BLISS REFERENCE MANUAL

A Basic Language for Implementation of System Software for the PDP-10

W. A. Wulf
D. Russell
A. N. Habermann
C. Geschke
J. Apperson
D. Wile

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

January 15, 1970

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

PREFACE

This manual is a definitive description of the BLISS language as implemented for the PDP-10. BLISS is a language specifically designed for writing software systems such as compilers and operating systems for the PDP-10. While much of the language is relatively "machine independent" and could be implemented on another machine, the PDP-10 was always present in our minds during the design, and as a result BLISS can be implemented very efficiently on the 10. This is probably not true for other machines.

We refer to BLISS as an "implementation language". This phrase has become quite popular lately, but apparently does not have a uniform meaning. Hence it is worthwhile to explain what we mean by the phrase and consequently what our objectives were in the language's design. To us the phrase "implementation language" connotes a higher level language suitable for writing production software; a truly successful implementation language would completely remove the need and/or desire to write in assembly language. Furthermore, to us, an implementation language need not be machine independent--in fact, for reasons of efficiency, it is unlikely to be.

Many reasons have been advanced for the use of a higher level language for implementing software. One of the most often mentioned is that of speeding up its production. This will undoubtedly occur, but it is one of the less important benefits, except insofar as it permits fewer, and better programmers to be used. Far more important, we believe, are the benefits of documentation, clarity, correctness and modifiability. These were the most important goals in the design of BLISS.

Some people, when discussing the subject of implementation languages, have suggested that one of the existing languages, such as PL/I, or at most

a derivative of one, should be used; they argue that there is already a proliferation of languages, so why add another. The only rational excuse for the creation of yet another new language is that existing languages are unsuitable for the specific applications in mind. In the sense that all languages are sufficient to model a Turing machine, any of the existing languages, LISP for example, would be adequate as an implementation language. However, this does not imply that each of these languages would be equally convenient. For example, FORTRAN can be used to write list processing programs, but the lack of recursion coupled with the requirement that the programmer code his own primitive list manipulations and storage control makes FORTRAN vastly inferior to, say, LISP for this type of programming.

What, then, are the characteristics of systems programming which should be reflected in a language especially suited for the purpose? Ignoring machine dependent features (such as a specific interrupt structure) and recognizing that all differences in such programming characteristics are only ones of degree, three features of systems programming stand out:

1. Data structures. In no other type of programming does the variety of data structures nor the diversity of optimal representations occur.
2. Control structures. Parallelism and time are intrinsic parts of the programming system problem.*
3. Frequently, systems programs cannot presume the existence of large support routines (for dynamic storage allocation, for example).

---

* Of course, parallelism and time are intrinsic to real time programming as well.

These are the principal characteristics which the design of BLISS attempts to address. For example, taking point (3), the language was designed in such a way that no system support is presumed or needed, even though, for example, dynamic storage allocation is provided. Thus, code generated by the compiler can be executed directly on a "bare" machine. Another example, taking point (1), is the data structure definition facility. BLISS contains no implicit data structures (and hence no presumed representations for structures), but rather provides a method for defining a representation by giving the explicit accessing algorithm.

One final point before proceeding with the description of the language--namely, the method of syntax specification. The syntax is given in BNF, for example

escapeexpression → EXITBLOCK escapeexpression | EXITLOOP escapeexpression

escapeexpression → | e

where: (1) lower case words are metalinguistic variables, and (2) the 'empty' construct is represented by a blank (as in the first alternative of the second rule above).

TABLE OF CONTENTS

I. LANGUAGE DEFINITION

1.1 Modules

A module is a program element which may be compiled independently of other elements and subsequently loaded with them to form a complete program.

module → block

A module may request access to other modules' variables and functions by declaring their names in EXTERNAL declarations. A module permits general use of its own variables and ROUTINEs by means of GLOBAL declarations. These lines of communication between modules are linked by the loader prior to execution. A complete program consists of an ordered set of compiled modules linked by the loader.

1.2 Blocks and Comments

A block is an arbitrary number of declarations followed by an arbitrary number of expressions all separated by semicolons and enclosed in a matching begin-end pair.

block → begin declarations compoundexpression end

declarations → |declaration;|declarations; declaration;

compoundexpression → |e| e; compoundexpression

begin → BEGIN

end → END

comment → | ! restofline endoflinesymbol|% stringwithnopercent %

Comments may be enclosed between the symbol ! and the end of the line on which the ! appears. However, a ! may appear in the quoted string of a literal, or between two % symbols, without being considered the beginning of a comment. Likewise, a % enclosed within quotes will be considered part of a string.

As in Algol the block indicates the lexical scope of the names declared at its head. However, in contrast to Algol, there is an exception. The names of GLOBAL variables and ROUTINEs have a scope beyond the block and although they are declared within the module, the effect, for a module citing them in an EXTERNAL declaration, is as if they were declared in the current block.

1.3 Literals

The basic data element is a PDP-10 36 bit word. However, the hardware provides the capability of pointing to an arbitrary contiguous field within a word and so a 36 bit word may be regarded as a special case of the "partial word". Literals are normally converted to a single word.

literal → number | quotedstring

number → decimal | octal

decimal → digit | decimal digit

octal → # oit | octal oit

digit → 0|1|2 --- |9

oit → 0|1|2 --- |7

Numbers (unsigned integers) are converted to binary modulo 2^{36} residue -2^{35}. The binary number is 2's complement and is signed. Octal constants are prefixed by the sharp sign, #.

quotedstring → leftadjustedstring | rightadjustedstring

leftadjustedstring → 'string'

rightadjustedstring → "string"

Quoted-string literals may be used to specify bit patterns corresponding to the 7-bit ASCII code for visable graphic characters on the external I/O media. Two types of single-word strings are provided for left or right justification of the string within a word. Normally quoted strings are limited to five characters and the unused bit positions are filled with zeroes. In OWN and GLOBAL declarations, the namesizevalue (see later material) may be of the form

namesizevalue → name ← (quotedstring)

In this special case, if the string is a leftadjustedstring, the string may be of arbitrary length and is bitten off in five character hunks and placed in successive words. The last word is leftadjusted and filled with trailing zero bits. The number of words so filled is such that there is at least one word with some zero (null) characters at the end.

Within a quoted string the quoting character is represented by two successive occurrences of that character.

1.4 Names

Syntactically an identifier, or name, is composed of a sequence of letters and/or digits, the first of which must be a letter. Certain names are reserved as delimiters, see Appendix A. Semantically the occurrence of a name is exactly equivalent to the occurrence of a pointer to the named item. The term "pointer" will take on special connotation later with respect to contiguous sub-fields (bytes) within a word; however, for the present discussion the term may be equated with "address". This interpretation of name is uniform throughout the language and there is no distinction between left and right hand values. Contrast this with Algol where a name usually, but not always, means "contents of".

The pointer interpretation requires a "contents of" operator, and "." has been chosen. Thus .A means "contents of location A" and ..A means "contents of the location whose name is stored in location A". To illustrate the concept, consider the assignment expression

simpleexpression → p11 ← e

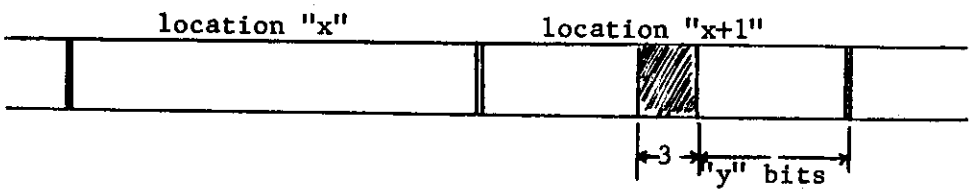
This means "store the value computed from e into the location whose pointer is the value of p11". (Further details are given in 2.2.) Thus the Algol statement "A := B" is written "A ← .B". It is impossible to express in Algol BLISS expressions such as: "A ← B", "A ← ..B", ".A ← .B", etc.

1.5 Pointers

As explained in 1.4, the value of a name is a pointer which names a location in memory. However, pointers are more general than mere addresses since they may name an arbitrary contiguous portion of a word, and may, further, involve index modification and indirect addressing. (For full details, the reader should refer to the PDP-10 System Reference Manual.) The most general form of pointer specifies five quantities; an example is $\epsilon_0 <\epsilon_1,\epsilon_2,\epsilon_3,\epsilon_4>$, where ϵ_0 is computed modulo 2^{18} and forms the base word address (Y field); ϵ_1,ϵ_2, are computed modulo 2^6 and form the position, size fields respectively (P, S fields); ϵ_3 is computed modulo 2^4 and forms the index field (X field); ϵ_4 is computed modulo 2 and forms the indirect address bit (I field). Each of $\epsilon_1,\epsilon_2,\epsilon_3,\epsilon_4$ may optionally be omitted, in which case a default value is supplied. $\epsilon_1,\epsilon_3,\epsilon_4$ have defaults of 0, but ϵ_2 has the default of 36. Thus, for example, the expression

$$(x+1)<.y,3>$$

defines a three bit field in the first location beyond x. The position of this three bit field is ".y" bits from the right end of the word.



1.6 The "contents of" Operators

The interpretation placed on identifiers in Bliss coupled with the dot operator discussed earlier allow a programmer direct access to, and control over, fields within words, to pointers to such fields which are themselves stored within memory, to chains of such pointers, etc. Two additional "contents of" operations besides the dot are provided which are more efficient in certain cases, but which are defined in terms of the dot and pointer operations. These operators are @ and \, and are defined by the following (where t is a temporary):

$$@\in \equiv .\in < 0, 36, 0, 0 >$$
$$\backslash\in \equiv .(t \leftarrow \in) < 0, 36, .t < 18, 4 >, .t < 22, 1 >>$$

Thus, both @∈ and \∈ specify a full 36 bit value. @∈ uses only the rightmost 18 bits of ∈ as the absolute address from which to fetch the value. \∈ interprets the rightmost 23 bits of ∈ as an indirect bit, index register field and base address. Whichever form is used, the compiler attempts to optimize the code produced; thus, for example, identical code is produced for .x, @x, and \x, if they occur in an expression.

Suppose that the assignment "X ← Y < 3, 15, R1, 0>;" has been executed, that is a pointer has been stored in X (that pointer has P=3, S=15, X=R1, I=0), and further that register R1 contains two. Now:

(1) Z ← .X stores the value of X, i.e.,the pointer, into Z

(2) Z ← ..X stores the value of the fifteen bit field (which ends three bits from the right) on the second word following Y into Z

(3) Z ← @ .X stores the value of Y into Z

(4) Z ← \ .X stores the value of the second word following Y into Z

(5) .X ← 5 stores 5 into the relevant fifteen bit field of the second word following Y

(6) @ X ← 5 stores 5 into Y

(7) \X ← 5 stores 5 into the second word following Y

2.1 Expressions

Every executable form in the BLISS language (that is, every form except the declarations) computes a value. Thus all commands are expressions and there are no "statements" in the sense of Algol or Fortran. In the syntax description e is used as an abbreviation for expression.

e → simpleexpression | controlexpression

2.2 Simple Expressions

The semantics of simpleexpressions is most easily described in terms of the relative precedence of a set of operators, but readers should also refer to the BNF-like description in 4.1. The precedence number used below should be viewed as an ordinal, so that 1 means first and 2 second in precedence. In the following table the letter ϵ has been used to denote an actual expression of the appropriate syntactic type, see 4.1.

Precedence	Example	Semantics
1	(compoundexpression)	The component expressions are evaluated from left to right and the final value is that of the last component expression.
1	block	
1	$\epsilon_0(\epsilon_1,\epsilon_2,\ldots,\epsilon_n)$	A function call, see 3.4.
1	name$[\epsilon_1,\epsilon_2,\ldots,\epsilon_n]$	A structure access, see 3.5.
1	name	A pointer to the named item, see 1.4.
1	literal	Value of the converted literal, see 1.3.
2	ϵ<pointer parameters>	A partial word pointer, see 1.5.
3	.ϵ	Value (possibly partial word) pointed at by ϵ.
3	@ϵ	Equivalent to .ϵ<0.36.0.0>.
3	\ϵ	Equivalent to .(t←ϵ)< 0,36,.t< 18,4>, .t<22,1>>.
4	$\epsilon_1\uparrow\epsilon_2$	ϵ_1 shifted logically by ϵ_2 bits; left if ϵ_2 positive; right if ϵ_2 negative. (Shifts are modulo 256.)
5	ϵ*ϵ	Product of ϵ's.
5	ϵ_1/ϵ_2	ϵ_1 divided by ϵ_2.
5	ϵ_1 MOD ϵ_2	ϵ_1 modulo ϵ_2.
6	-ϵ	Negative of ϵ.
6	ϵ+ϵ	Sum of ϵ's.
6	ϵ_1-ϵ_2	Difference between ϵ_1 and ϵ_2.

[Note all arithmetic is carried out modulo 2^{36} with a residue of -2^{35}. All arithmetic is integer; if floating point arithmetic is introduced it will be by means of special operators, viz.,FMP, FDV, FNE, FAD, FSU.]

Precedence	Example	Semantics
7	ϵ_1 EQL ϵ_2	$\epsilon_1 = \epsilon_2$
7	ϵ_1 NEQ ϵ_2	$\epsilon_1 \neq \epsilon_2$
7	ϵ_1 LSS ϵ_2	$\epsilon_1 < \epsilon_2$
7	ϵ_1 LEQ ϵ_2	$\epsilon_1 \leq \epsilon_2$
7	ϵ_1 GTR ϵ_2	$\epsilon_1 > \epsilon_2$
7	ϵ_1 GEQ ϵ_2	$\epsilon_1 \geq \epsilon_2$

[Truth is represented by 1, falsity by 0.]

8	NOT ϵ	bitwise complement of ϵ
9	ϵ AND ϵ	bitwise and of ϵ's
10	ϵ OR ϵ	bitwise inclusive or of ϵ's
11	ϵ XOR ϵ	bitwise exclusive or of ϵ's
11	ϵ EQV ϵ	bitwise equivalence of ϵ's
12	$\epsilon_1 \leftarrow \epsilon_2$	The value of this expression is identical to that of ϵ_2, but as a side effect this value is stored into the partial word pointed to by ϵ_1; with associative use of $\leftarrow$, the assignments are executed from right to left: thus $\epsilon_1 \leftarrow \epsilon_2 \leftarrow \epsilon_3$ means $\epsilon_1 \leftarrow (\epsilon_2 \leftarrow \epsilon_3)$. However, in general, there is no guarantee regarding the order in which a simpleexpression is evaluated other than that provided by precedence and nesting: thus (R ← 2; @ R * (R ← 3)) may evaluate to 6 or 9.

The reader should refer to the PDP-10 reference manual for a complete definition of the arithmetic operators under various special input value conditions.

2.3.1 Control Expressions

The controlexpressions provide sequencing control over the execution of his program; there are five forms:

controlexpression → conditionalexpression | loopexpression | parallelexpression | escapeexpression | coroutineexpression

The general goto statement has deliberately been omitted from the language to improve readability and structuring of programs.

2.3.2 Conditional Expressions

conditionalexpression → IF e_1 THEN e_2 ELSE e_3

e_1 is computed and the resulting value is tested. If it is odd*, then e_2 is evaluated to provide the value of the conditional expression, otherwise e_3 is evaluated.

conditionalexpression → IF e_1 THEN e_2

This form is equivalent to the IF-THEN-ELSE form with 0 replacing e_3. However, it does introduce the "dangling else" ambiguity. This is resolved by matching each ELSE to the most recent unmatched THEN as the conditional expression is scanned from left to right.

---

* Only the least significant bit of e_1 is tested; a zero bit is interpreted as false and a one bit as true. Thus any odd integer value is interpreted as true and any even value as false.

2.3.3 Loop Expressions

The value of each of the six loop expressions is -1, except when an EXITLOOP is used, see 2.3.4.

loopexpression → WHILE e_1 DO e_2

The e_1 is computed and the resulting value is tested. If it is odd, then e_2 is computed and the complete loopexpression is recomputed; if it is even, then the loopexpression evaluation is complete.

loopexpression → UNTIL e_3 DO e_2

This form is equivalent to the WHILE-DO form except that e_1 is replaced by NOT(e_3).

loopexpression → DO e_2 WHILE e_1

The expressions e_2,e_1 are computed in that sequence. The value resulting from e_1 is tested: if it is odd, then the complete loop expression is recomputed; if it is even, then the loopexpression evaluation is complete.

loopexpression → DO e_2 UNTIL e_3

This form is equivalent to the DO-WHILE form except that e_1 is replaced by NOT(e_3).

loopexpression → INCR name FROM e_1 TO e_2 BY e_3 DO e_4

This is a simplified form of the Algol 68 for-loop. The "name" is declared to be a REGISTER or a LOCAL for the scope of the loop. The expression e_1 is computed and stored in name. The expressions e_2 and e_3 are computed and stored in unnamed local memory which for explanation purposes we shall name U_2 and U_3. Any of the phrases "FROM e_1" "TO e_2" or "BY e_3" may be omitted--

in which case default values of $e_1 = 0$, $e_2 = 0$, $e_3 = 1$ are supplied. The following loopexpression is then executed:

DO (e_3; name ← .name+U_3) UNTIL .name GTR .U_2;

The final form of a loopexpression is:

loopexpression → DECR name FROM e_1 TO e_2 BY e_3 DO e_4

This is equivalent to the INCR-FROM-TO-BY-DO form except that the final loop is replaced by

DO (e_3; name ← .name-U_3) UNTIL .name LSS .U_2;

2.3.4 Escape Expressions

The various forms of escapeexpressions permit control to leave its current environment. They are intended for those circumstances when other controlexpressions would have to be contorted to achieve the desired effect.

escapeexpression → environment level escapevalue | RETURN escapevalue

environment → EXIT | EXITBLOCK | EXITCOMPOUND | EXITLOOP | EXITCONDITIONAL

levels → | [e]

escapevalue → | e

Each of these expressions conveys to its new environment a value, say $\in$, obtained by evaluating the escapevalue, which may optionally be omitted implying $\in = 0$. The levels field, which must evaluate to a constant, say n, at compile time, determines the number of levels of the specified control environment to be exited; the levels field may optionally be omitted in which case one level is implied. The maximum number of levels which may be exited in this way is limited by the current function (routine) body or the outermost block.

RETURN	terminates the current function, or routine, with value $\in$.
EXITBLOCK	terminates the innermost n (where n is the value of the "levels" field) blocks, yielding a value of $\in$ for the outermost one exited.
EXITCOMPOUND	terminates the innermost n compound expressions, yielding a value of $\in$ for the outermost one exited.
EXITLOOP	terminates the innermost n loop expressions, yielding a value of $\in$ for the outermost one exited.
EXITCOND	terminates the innermost n conditional expressions, yielding a value of $\in$ for the outermost one exited.
EXIT	terminates the innermost n control scopes (whether blocks, compounds, conditionals, or loops with $\in$ as the value of the outermost.

2.3.5 Parallel Expressions

parallelexpression → SET expressionset TES

expressionset → |e|; expressionset | e ; expressionset

When a compoundexpression is enclosed between a pair of parentheses or a BEGIN-END pair, its component e's are evaluated sequentially from left to right, the rightmost providing the final value. However, when an expressionset is enclosed between a SET-TES pair the order of evaluation is undefined which implies that the constituent e's may even be evaluated in whatever order the compiler finds convenient, and possibly even in parallel. The value of the SET-TES expression is that of the last e to be evaluated at execution time. Thus, unless there is only one e the value is unpredictable.

The n expressions should be sufficiently restrictive that the final state is independent of the actual order in which the component expressions are evaluated. An escape expression is illegal where its use would imply escaping from the SET-TES environment. Another form of parallelexpression is:

parallelexpression → CASE elist OF SET expressionset TES

Let us suppose that the actual e's within the elist are $\epsilon_1,\epsilon_2,\ldots,\epsilon_m$ and that the actual expressions within the expressionset are $\Pi_0;\Pi_1;,,,;\Pi_n$. Then the expressions $\{\Pi_{\epsilon_i} \mid i = 1,2,\ldots,m\}$ are executed as if sandwiched between a SET and a TES. The number of selected expressions is m. For m=1 the case expression is sequential with no forking and has a predicatable value (that of the selected expression).

parallelexpression → SELECT elist OF NSET nexpressionset TESN

nexpressionset → | ne | ne; nexpressionset

ne → e:e

This form is somewhat similar to the case expression except that the expressions in the nexpressionset are not thought of as being sequentially numbered--instead each expression in the nexpressionset is tagged with an "activation" expression. Suppose we have the following select expression

SELECT ϵ_1, ϵ_2, ϵ_3 OF NSET ϵ_4: ϵ_5; ϵ_6: ϵ_7; ϵ_8: ϵ_9; ϵ_{10}: ϵ_{11} TESN

then the execution proceeds as follows: first ϵ_1, ϵ_2, ϵ_3 are evaluated, then ϵ_4, ϵ_6, ϵ_8 and ϵ_{10} are evaluated; correspondingly ϵ_5 is evaluated if and only if ϵ_4 is equal to one of ϵ_1, ϵ_2, or ϵ_3. Similarly ϵ_7 is evaluated if an only if ϵ_6 is equal to one of ϵ_1, ϵ_2, or ϵ_3, etc. As with the case expression, the order of evaluation of the nset elements is not defined and the value of the entire expression is that of the last one to be executed at execution time. Thus, the value of the complete select expression is uniquely determined only in the case that the elist contains precisely one element.

An escape expression is illegal where its execution would imply escape from an NSET-TESN environment.

2.3.6 Co-routine Expressions

The body of a function or routine may be activated as a co-routine and/or asynchronous process; the additional syntax is

coroutineexpression → CREATE e_1 (elist) AT e_2 LENGTH e_3 THEN e_4
EXCHJ (e_4, e_5)

The effect of a 'create' expression is to create a context, that is an independent stack, for the routine (function) named by e_1, with parameters specified by the elist, at the location whose address is specified by e_2 and of size e_3 words. Control then passes to the statements following the 'create'. When two or more such contexts have been established, control may be passed from any one to any other by executing an exchange-jump, EXCHJ (e_4, e_5)* where the value of e_4 must be the stack base, e_2, of a previous 'create' expression. The value of e_5 is made available to the called routine as the value of its own EXCHJ which caused control to pass out of that routine. Thus the value of the EXCHJ operation is defined dynamically by the co-routine which at some later time re-activates execution of the current co-routine.**

Should a process, the body of which is necessarily that of a function (or routine), execute a 'return', either explicitly or implicitly, the expression e_4 (following the 'then' in the 'create' expression of the creating process) is executed in the context of the <u>created</u> process. The normal responsibilities of e_4 include making the stack space used for the created context available for other uses and performing an EXCHJ to some other process.

The facilities described above, namely 'create' and 'exchj', are adequate either for use directly as co-routine linkages or for use as primitives in constructing more sophisticated co-routine facilities with macros

*Note that the 1st EXCHJ to a newly created process causes control to enter from its head with actual parameters as set up by the CREATE.

**The value e_5 is not available to the called routine on the 1st EXCHJ to it.

and/or procedures. It should be noted in the context that if the created processes are functions (rather than routines) the resulting processes continue to have access to lexically global variables which may be local to an embracing function (access to lexically local variables which have been declared 'own' is available in either case). In such a case the resulting structure is a stack tree in which all segments of the tree below the lexical level of the (function) process are available to it.

Two additional complexities are added if the create and exchj are to be used for asynchronous, and possibly parallel, execution of processes. One is synchronization, by which we man a mechanism by which a process can coordinate its execution with that of one or more others. A typical example of the need for synchronization occurs when two processes, independently update a common data base, and each must be sure that the entire updating process is complete before any other process attempts to use the data base. The second complexity arises in connection with interrupts, and in particular from the fact that certain operations must not be interrupted (some exchj operations for example). It is possible that certain situations require synchronization mechanisms but do not need to be concerned about the interrupt problem--as for example, a user program with asynchronous processes, which is 'blind' to interrupts, and which some monitor systems view as a single 'job'.

The nature of "appropriate" synchronization primitives and mechanisms for temporarily blinding the processor to interrupts (or interrupts in a certain class) are highly dependent upon the nature of the processes being used and the operating system, or lack of one, underlying the Bliss program. As a consequence, no syntax for dealing with either problem is included in

the language; in any case, the amount of code necessary for these facilities is quite small.

The co-routine user is well advised to read and understand the material on the run-time representation of Bliss programs contained in section IV.

3.1 Declarations

All declarations, except MAP, introduce names each of which is unique to the block in which the declaration appears. Except with STRUCTURE and MACRO declarations, the name introduced has a pointer bound to it.

3.2 Memory Allocation

There are four classes of declaration which allocate memory space.

declaration → LOCAL namesizelist |
REGISTER namesizelist |
OWN namesizevaluelist |
GLOBAL namesizevaluelist

namesizelist → namesize | namesizelist, namesize

namesize → $name_1$ | $name_2$ [e_2]

namesizevaluelist → namesizevalue | namesizevaluelist, namesizevalue

namesizevalue → namesize | $name_1$ ← e_1 | $name_2$ [e_2] ← (valuelist) |
$name_3$ ← quotedstring

valuelist → value | valuelist, value

value → e_4 | e_5 (valuelist)

With LOCAL and REGISTER every name in the namesize list is declared to have a scope coincident with the current block. For every incarnation of the block at run time (including parallel incarnations of the same routine via the 'create' mechanism) one word of memory is allocated for $name_1$ and e_2 words of memory are allocated for $name_2$. The memory space for a particular incarnation is released at the corresponding block exit. The names have as value the pointer to the first (or only) word of memory allocated. The contents of the allocated memory is undefined and should not be presumed. The memory space is taken from core (LOCAL) or the high speed registers (REGISTER) as specified. Also, e_2 is restricted to an expression which is calculable at compile time. Registers must be used sparingly since less than the full 16 will be available for general use.

With OWN and GLOBAL, for every name in the namesizevalue list one word of memory is allocated for $name_1$ and e_2 words are allocated for $name_2$. The memory space is taken from core at compile time and survives for the complete run. The names have as value the pointer to the first (or only) word of memory allocated. The content of word $name_1$ may be initialized at compile time to e_1. The contents of the e_2 words commencing at $name_2$ may be initialized to the values in a valuelist. Whereas e_4 is a single value, there are e_5 occurrences of its ensuing valuelist. The expressions e_1, e_2, e_4, e_5 are restricted to being calculable at compile time. Enough words are allocated for $name_3$ to store the quoted string. The scope of an OWN name is that of the block in which it is declared and of a GLOBAL name is that of the outermost block of the final program. GLOBAL names are made available to another module by citation in that module's EXTERNAL list. Note that co-executing incarnations of the same block, whether invoked as a recursive subroutine or as a co-routine (or both) refer to the same location if that location was declared by OWN or GLOBAL declarations.

3.3 Module Communication

There are two declarations by means of which modules may access names of another module. The GLOBAL declaration has already been discussed (3.2).

declaration → EXTERNAL namelist

namelist → name | namelist, name

Each name in the namelist of an EXTERNAL declaration must be defined by a GLOBAL declaration in another module to which the current module will be linked before execution. The EXTERNAL declaration makes these names known to the current block of the current module via the loader.

3.4 Functions

```
declaration → FUNCTION name (namelist) = e |
              FUNCTION name = e |
              ROUTINE name(namelist) = e |
              ROUTINE name = e
```

The FUNCTION and ROUTINE declarations define the name to be that of a potentially recursive and re-entrant function whose value is the expression e. The syntax of a normal subroutine-like function call is

```
p1 → p1 (elist) | p1 ( )
elist → e | elist, e
```

where p1 is a primary expression. Clearly, p1 must evaluate to a name which has been declared as a FUNCTION or ROUTINE either at compile time or at run time. The names in the namelist of the declaration define (lexically local) the names of formal parameters whose actual values on each incarnation are determined by the elist at the call site. All parameters are implicitly Algol "call-by-value"; but notice that call-by-reference is achieved by simply presenting pointer values at the call site. Parentheses are required at the call site even for a ROUTINE or a FUNCTION with no formal parameters since the name on its own is simply a pointer to the function or routine. Extra actual parameters above the number mentioned in the namelist of the function (or routine) declaration are always allowed; however, too few actual parameters can cause erroneous results at run time. A ROUTINE differs from a FUNCTION in having an abbreviated and hence faster prolog. Restriction: a routine may not refer directly to local variables declared outside it, nor may it call a FUNCTION.

declaration → GLOBAL ROUTINE name (namelist) = e |
GLOBAL ROUTINE name = e

A ROUTINE name is like an OWN name in that its scope is limited to the block in which it is declared and its value is already initialized at block entry. The prefix GLOBAL changes the scope of the ROUTINE to that of the outer block of the program enveloping all the modules. Note that this inhibits a GLOBAL ROUTINE from access to REGISTER names declared outside it. This is in addition to the other limitations of ROUTINES cited on the previous page.

Functions and routines may also be activated as co-routines and/or asynchronous processes, and indeed, the body of a single function may be used in any or all of these modes simultaneously. (See 2.3.6.)

declaration → EXTERNAL nameparlist |
FORWARD nameparlist

nameparlist → namepar | nameparlist, namepar

namepar → name (e)

EXTERNAL and FORWARD each tell the compiler how many parameters, given by e*, are expected by an undeclared function (or routine), name. FORWARD is for functions (or routines) declared later in the current block and EXTERNAL is for routines from another module. The compiler permits the number of actual parameters in a function (or routine) call to be greater than or equal to the number of formals declared.

*Clearly e must evaluate to a constant at compile time.

3.5 Structures

Structure declarations serve to define data structures by giving an explicit algorithm for the "indexing rule" associated with that class of structures.

declaration → STRUCTURE name [namelist] = e

This declaration introduces name as a new "structure class" by which specific data names will be mapped in a MAP declaration. The names in the namelist are formal parameter names which positionally correlate with actual parameters in the usual manner. In addition, the structure class name is used to denote the 0th formal parameter which will correlate with the name (base address) of the data space used at the call site. The syntax of a structure access is

p1 → name [elist]

Before describing the meaning of this we must examine the MAP declaration.

declaration → MAP name: namelist

Here name must be defined by a STRUCTURE declaration, and the names in the namelist must be defined as memory space. The MAP declaration permits the memory space to be accessed by the indexing rule specified by the STRUCTURE declaration. In the following example, TRI may be accessed as a symmetric matrix although only the lower triangle is stored.

```
OWN TRI[5*6/2],DOPE[5] ← (0,1,3,6,10);
STRUCTURE VEC[I] = (.VEC-1+.I);
STRUCTURE SYM[I,J] = (.SYM-1+(IF.I GTR.J THEN.DOPE[.I]+.J ELSE
                                              .DOPE[.J]+.I));
MAP VEC:DOPE;
MAP SYM:TRI;
```

A given memory space may be accessed in more than one way by binding alias names to it and mapping a different structure on each alias.

declaration → BIND equivalencelist

equivalencelist → equivalence | equivalencelist, equivalence

equivalence → name = e

Referring to the previous example we could access TRI linearly by means of the alias LIN, thus:

```
BIND  LIN = TRI;
MAP   VEC : LIN;
```

Notice that the value to which a name may be bound need not evaluate at compile time but may be determined at execution time. For example, in the following code this feature is used to effect a row interchange within a matrix.

```
BEGIN
STRUCTURE ARY2[I,J] = .ARY2 + (.I-1)*10 + (.J-1);
STRUCTURE ARY1[I] = .ARY1 + .I-1;
OWN X[100];
MAP ARY2:X;
...
BEGIN BIND XX1 = X[.K,1], XX2 = [.n,1]; MAP ARY1:XX1,XX2; REGISTER T;
  INCR I FROM 1 TO 10 DO
    (T ← XX1[.I]; XX1[.I] ← XX2[.I]; XX2[.I] ← T);
END;
...
End;
```

3.6 Macros

Macro expansion takes place during compilation after lexical analysis but before syntactic analysis. The range of a macrocall is sufficiently general that it cannot be described in simple BNF. The only restrictions on the positioning of a macrocall are that it may not appear as part of a literal, name or reserved word, nor may it appear until lexically after the corresponding declaration, so that the recursive macros are impossible.

declaration → MACRO definitionlist

definitionlist → definition | definitionlist, definition

definition → $name_1$ (namelist) = $matchedstring_1$ $ |
$name_2$ = $matchedstring_2$ $

The matchedstring may be an arbitrary string of atoms of the language, except that any occurrences of "MACRO" and "$;" must be as nested ordered pairs.

macrocall → $name_1$ (stringlist) | $name_2$

stringlist → string | stringlist, string

Each string in the stringlist may contain any symbol other than a comma. For the simple macro without parameters, expansion consists of simply replacing every appearance of $name_2$ for its scope by $matchedstring_2$. For the parameterized macro, every occurrence in the $matchedstring_1$ of each name in the namelist is replaced by the corresponding string in the stringlist. The modified (expanded) string then replaces the call in the program. After expansion the input scanner is left pointing at the first symbol of the expanded string so that macrocalls may be nested. Where a macrocall appears in the matchedstring it is not expanded at the declaration but at call sites of the enclosing macro.

Macros may be used to provide names to bit fields so as to improve readability.

```
MACRO EXPONENT = 27,8 $;
MACRO MANTISSA = 0,27 $;
MACRO SIGN = 35,1 $;
LOCAL X;
X <SIGN> ← 0; X <EXPONENT> ← 27; X <MANTISSA> ← .I;
```

Macros may be used to extend the syntax in a limited way.

```
MACRO NEG = 0 GTR $;
MACRO UNLESS(X) = IF NOT(X) $;
```

Macros may be used to effect in-line coding of a function.

```
MACRO ABS(X) = BEGIN REGISTER TEMP;
               IF NEG(TEMP ← X) THEN -.TEMP ELSE .TEMP END $;
! HERE THE ACTUAL PARAMETER SUBSTITUTED FOR X MAY NOT INCLUDE THE
!                                                    NAME TEMP.
```

II. SPECIAL LANGUAGE FEATURES

The previous chapter describes the basic features of the BLISS language. In this chapter we describe additional features which are highly machine and implementation dependent.

1.1 Special Functions

A number of features have been added to the basic BLISS language which allow greater access to the PDP-10 hardware features. These features have the syntactic form of function calls and are thus referred to as "special functions". Code for special functions is always generated in line.

1.2 Character Manipulation Functions

Ten functions have been specified to facilitate character manipulation operations. They are:

scann (ap)	copynn (ap_1, ap_2)
scani (ap)	copyni (ap_1, ap_2)
replacen (ap, $\in$)	copyin (ap_1, ap_2)
replacei (ap, $\in$)	copyii (ap_1, ap_2)
incp (ap)	
decp (ap)	

For each of these $\in$ is an arbitrary expression, and ap is an expression whose value is a pointer to a pointer. The second of these pointers is assumed to point to a character in a string.

scann (ap)	is a function whose value is the character from the string.
scani (ap)	is like scann except that, as a side effect, the string pointer is set to point at the next character of the string <u>before</u> the character is scanned.
replacen (ap, $\in$)	is a function whose value is $\in$ and which, as a side effect, replaces the string character by $\in$.
replacei (ap, $\in$)	is similar to replacen except that the string pointer is set to point at the next character of the string <u>before</u> the value of $\in$ is stored.
copynn (ap_1, ap_2) copyni (ap_1, ap_2) copyin (ap_1, ap_2) copyii (ap_1, ap_2)	these functions are similar in that they each effect a copy of one character from a source string (pointed at by $.ap_1$) to a destination string (pointed at by $.ap_2$) and have as value the character copied. They differ in that copynn advances neither pointer, while copyni advances $.ap_2$, copyin advances $.ap_1$, and copyii advances both. In each case the pointer is advanced <u>before</u> the copy is effected.
incp (ap)	advances .ap to the next character
decp (ap)	resets .ap to point at the previous character of the string.

Suppose that a string (of 7 bit ASCII characters) is stored in memory beginning at location S. The string is terminated by a null (zero) character. The following skeletal code will transform it into a 6-bit string with blanks deleted:

```
begin
register p7, p6, c;
p7 ← (s-1) <1, 7>; p6 ← (s-1) <0,6>;
while (c ← scani (p7)) neq 0 do
      if .c neq " " then replacei (p6, .c);
...
end;
```

1.3 Machine Language

It is possible to insert PDP-10 machine language instructions into a Bliss program in the syntactic form of a special function

$$op\ (\epsilon_1,\ \epsilon_2,\ \epsilon_3,\ \epsilon_4)$$

where

- op is one of the PDP-10 machine language mnemonics (see table below).
- ϵ_1 is an expression whose least significant 4 bits will become the accumulator (A) field of the compiled instruction. This expression must yield a value at compile time of a declared register name or a literal.
- ϵ_2 is an expression whose least significant 18 bits will become the address (Y) field of the compiled instruction.
- ϵ_3 is an expression whose least significant 4 bits will become the index (X) field of the compiled instruction.
- ϵ_4 is an expression whose least significant bit will become the indirect (I) bit of the compiled instruction.

(A table of machine language instruction mnemonics follows. Defaults for ϵ_1-ϵ_4 are 0.)

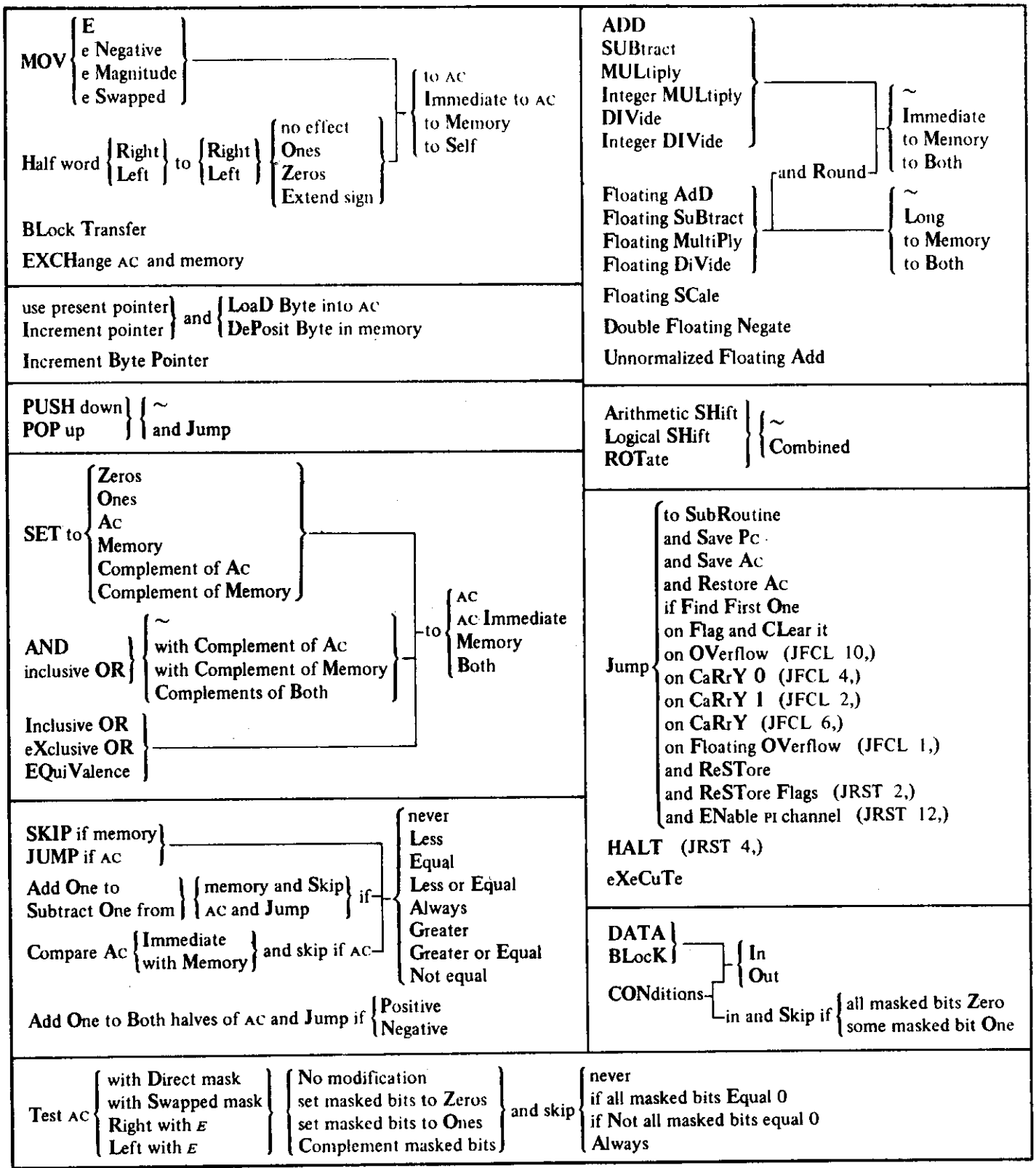
The 'value' of these machine language instructions is uniformly taken to be the contents of the register specified in the accumulator (A) field of the instruction. (This makes little sense in a few cases, but was adopted for uniformity.)

In order for the compiler to conserve space during compilation, the mnemonics for the machine language operators are not normally preloaded into the symbol table. Therefore, in order to use this feature of the language, it is necessary for the programmer to include one of the following special declarations

```
declaration → MACHOP  mlist  | ALLMACHOP
mlist       → name = e | mlist , name = e
```

in the head of a block which embraces occurrences of these special functions. (Note: the e's in an mlist must be the actual values of the machine operation and must evaluate at compile time.) Symbol table space for these names is released when the block in which the declaration occurs is exited.

PDP-10 Instruction Mnemonic Table *



*Reproduced with permission of Digital Equipment Corporation from the PDP-10 Reference Handbook.

IV. RUN TIME REPRESENTATION OF PROGRAMS

1.1 Introduction

In order to make the fullest possible use of Bliss, it is important to understand the run-time environment in which Bliss programs run. The address space is occupied by various types of information:

(1) program

(2) constants

(3) static size variable areas (globals and owns)

(4) stacks

Programs are 'pure' (they do not modify themselves) therefore program and constant areas are placed in contiguous, write-protected regions and may be shared. Static variable storage and stack space are placed in readable/writable memory. The key to understanding the run-time environment in the stack configuration and register allocation is illustrated in Figure IV.1. Each process (co-routine) has its own stack configured as shown in IV.1.

1.2 The Stack and Functions

The first 17_{10} locations of each stack are reserved for state information (registers plus program counter) for a process when it is inactive. The use of these cells is explained more fully in 1.4. The configuration above these 17 state words depends upon the depth of nesting of function calls, but each such nested call involves a similar (not identical) use of the stack; Figure IV.1 illustrates a typical stack configuration after several nested functional calls. At a time when one of these functions is executing

(1) The S-register points to the highest assigned cell in the stack; the S-register is used to control the allocation of the stack area.

(2) The F-register points to the 'local base of stack'; below* the F-register are the parameters to the function and the return address. The stack cell actually pointed to by the F-register contains the previous value of the F-register at the time at which the current function was entered.

(3) The calling sequence which is used to enter a function (or routine) is

```
PUSH   S,p1          ;  push 1st parameter onto the stack
PUSH   S,p2          ;  push 2nd parameter onto the stack
 ...                 ...
PUSH   S,pn          ;  push nth parameter onto the stack
PUSHJ  S,FCN         ;  jump to the called function
SUB    S,[nooooon]   ;  delete the parameters
```

(4) Above the F-register are stored the "displays", $D_1 \ldots D_f$.

*'below' in the sense of decreasing address values.

One display is used for each lexical nesting of the declaration of the function which is currently executing. The value of the displays are the F-register values for the most recent recursive entries for the lexically embracing functions. The displays are needed and used to access variables global to the current functions but local to embracing functions. Such access is prohibited in routines, and consequently no displays are saved on a routine entry.

(5) Above the displays are saved any working registers which are destroyed by the execution of the function body. These registers are restored before the function exits.

(6) Any local variables in the function are stored on top of the saved registers. Space is acquired/deleted for locals on block entry/exit by simply adding/subtracting a constant to the S-register.

(7) An excessive number of declared registers, or the evaluation of an unbelievably complex expression may exhaust the available registers, forcing the area above the locals to be used for storing partial results of an expression evaluation.

(8) The V-register is used to return the value of the function or routine.

Figure IV.2 illustrates the code generated surrounding the body of a function. The code surrounding a routine body is identical with the exception that the displays are never saved.

Figure IV.1

Stack Structure and Registers for a Process

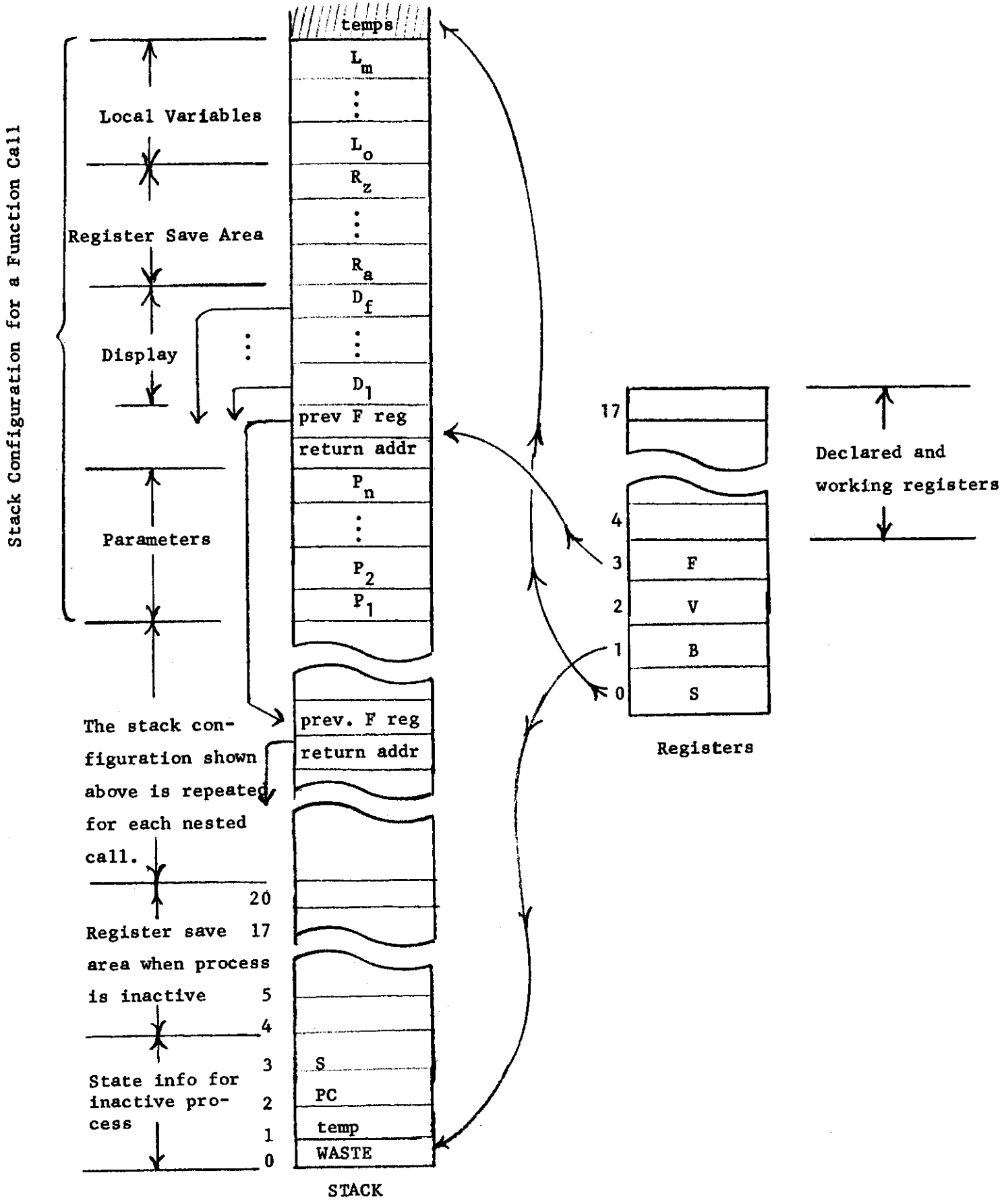


Figure IV.2

Function Prolog and Epilog

```
FCN:  PUSH   S,F          ;    save old F-register
      PUSH   S,1(F)       ;    copy display zero
      ...                 ...
      PUSH   S,f(F)       ;    copy display f
      HRRZ   F,S          ;    set up new F
      SUBI   F,f          ;    subtract no. displays
      PUSH   S,F          ;    new display created
      PUSH   S,Ra         ;    save register
      ...                 ...
      PUSH   S,Rz         ;    save register
      { BODY OF FUNCTION OR ROUTINE }
      POP    S,Rz         ;    restore register
      ...                 ...
      POP    S,Ra         ;    restore register
      SUB    S,[(f+1)00000(f+1)] ; eliminate displays
      POP    S,F
      POPJ   S,
```

Not Generated For Routines

Figure IV.3

Block Entry and Exit

```
BENTER:   MOVEM    R1,l+1(F)          ; save in-use working registers
          ...                         ...
          MOVEM    Rj,l+j(F)          ; save in-use working registers
          ADD      S ,[n00000n]       ; INCR S-register by no. locals in blk

BEXIT:    SUB      S,[(n+j)00000(n+j)] ; DECR S-register by no. locals in blk
                                       ; (note: in-use reg!s left in stack,
                                       ; re-loaded only when used)
```

1.3 Access to Variables

This section briefly indicates the mechanisms by which generated code accesses various types of variables (formals, owns and globals, locals, etc.) The exact addressing scheme used by the compiler in any particular case is highly dependent upon the context; however, the following material should aid in understanding the overall strategy.

(a) OWN and GLOBAL variables are accessed directly.

(b) Formal parameters of the current routine are accessed negatively with respect to the F-register. If the current routine has n formals, then the ith one is addressed by

$$(-n + i - 2)(F)$$

(c) Local variables of the current routine are accessed positively with respect to the F-register. To access the ith local cell, one uses

$$(i + d + r + 1)(F)$$

where d is the number of displays saved and r is the number of registers saved on function entry.

(d) Formal parameters and local variables which are not declared in the currently executing function are accessed through the display. The appropriate display is copied into one of the working registers then accessed by indexing through that register in a manner similar to that shown in (b) or (c) above.

1.4 Co-routine Creation and Calls

The two co-routine mechanisms are the 'create' and the 'exchj' expressions; an understanding of the latter is necessary to an understanding of the former--therefore, we shall describe it first.

Assume two processes P1 and P2 with respective stack bases β_1 and β_2. Further, assume P1 is active (P2 inactive) and executes

$$x \leftarrow \text{exchj } (\beta_2, v);$$

The following code is compiled in P1*

```
        PUSH      B,[β2]      ; store new stack base addr. in temp.
        MOVE(I)** V,v         ; parameter to β2 left in value register
        PUSHJ     B,EXCHRT    ; jump to routine to handle the exchange
```

where

```
EXCHRT: PUSH      B,S         ; save caller's S-register
        ADDI      B,1         ; set up destn for BLT, and test
        MOVE      S,B         ; copy B in preparation for BLT
        BLT       S,(17-3)(B) ; save caller's registers
        HRRZ      B,-2(B)     ; pick up new stack base
        HRLI      S, (B)      ; set up source for BLT
        HRRI      S,3         ; set up destination for BLT
        BLT       S,17        ; restore called program's registers
        MOVE      S, (B)      ; restore called program's stack ptr.
        JRSTF     @ (B)       ; jump to called program
```

The instructions generated for a 'create' simply establishes a stack configuration appropriate for a later EXCHJ. In particular, suppose a process executes

$$\text{CREATE } P_o(P_1, P_2, \ldots, P_n) \text{ AT } e_1 \text{ LENGTH } e_2 \text{ THEN } e_3$$

Then the following code is generated

** The exact form of this code depends upon the nature of the expression v.

* Note all numbers in code are octal.

```
        HRRZ(I)*    t1,e1       ; pick up the new stack base addr
        ADDI        t1,1        ; move past WASTE cell
        MOVE        t2,t1       ; make another copy for the BLT
        BLT         t2,20(t1)   ; save the registers
        MOVEN*      t2,e2       ; get negative length
        HRL         t1,t2       ; set length in stack pointer
        MOVE        t2,t1       ; be sure to have good copy of base addr.
        ADD         t1,[20000020]; bump pointer above save area
        PUSH        t1,p1       ;
        ...                     ; push parameters
        PUSH        t1,pn       ;
        PUSH        t1,[E3]     ; phoney return to e3
        MOVEM       t1,3(t2)    ; save S-register for new process
        MOVE(I)*    t1po        ; get entry point for new process
        MOVEM       t1,2(t2)    ; save entry point in state area
        JRST        ARNDIT      ; skip around e3 code
ETWO:   code for e2
ARNDIT:
```

*The form of the code obviously depends upon its actual form in the 'create' expression.

APPENDIX A: SYNTAX

```
module          → block
block           → begin declarations compoundexpression end
begin           → BEGIN
end             → END
comment         → |! restofline endoflinesymbol | % stringwithnopercent %
declarations    → | declaration; | declarations declaration;
declaration     → LOCAL namesizelist |
                  REGISTER namesizelist |
                  OWN namesizevaluelist |
                  GLOBAL namesizevaluelist |
                  EXTERNAL namelist |
                  FORWARD nameparlist |
                  FUNCTION name (namelist) = e |
                  FUNCTION name = e |
                  ROUTINE name (namelist) = e |
                  ROUTINE name = e |
                  GLOBAL ROUTINE name (namelist) = |
                  BIND equivalencelist |
                  STRUCTURE name [namelist] e |
                  MAP name:  namelist |

                  MACRO definitionlist
namesizelist    → namesize | namesizelist, namesize
namesize        → name | name [e]
```

```
namesizevaluelist  → namesizevalue | namesizevaluelist, namesizevalue
namesizevalue      → namesize | name ← e | name [e] ← (valuelist) |
                     name ← quotedstring
valuelist          → value | valuelist, value
value              → e | e (valuelist)
namelist           → name | namelist, name
nameparlist        → namepar | nameparlist, namepar
namepar            → name (e)
equivalencelist    → equivalence | equivalencelist, equivalence
equivalence        → name = e
definitionlist     → definition | definitionlist, definition
definition         → name (namelist) = matchedstring $ |
                     name = matchedstring $
compoundexpression → | e | e ; compoundexpression
e                  → controlexpression | simpleexpression
controlexpression  → conditionalexpression |
                     loopexpression |
                     escapeexpression |
                     parallelexpression |
                     coroutineexpression
conditionalexpression → IF e THEN e ELSE e |
                        IF e THEN e
loopexpression     → WHILE e DO e |
                     UNTIL e DO e |
                     DO e WHILE e |
                     DO e UNTIL e |
```

```
                        INCR name FROM e TO e BY ∈ DO e |

                        DECR name FROM e TO e BY ∈ DO e

escapeexpression    → environment levels escapevalue | RETURN escapevalue

levels              → | [e]

escapevalue         → | e

environment         → EXIT | EXITBLOCK | EXITCOMPOUND |
                      EXITLOOP | EXITCOND

parallelexpression  → SET expressionset TES |

                        CASE elist OF SET expressionset TES |

                        SELECT elist of NSET nexpressionset TESN

expressionset       → |e| ; expressionset | e ; expressionset

nexpressionset      → |ne| ne ; nexpressionset

elist               → e | elist, e

ne                  → e:e

coroutineexpression→ CREATE e (elist) AT e LENGTH e THEN e | EXCHJ (e,e)

simpleexpression    → p11 ← e | p11

p11                 → p10 | p11 XOR p10 | p11 EQV p10

p10                 → p9 | p10 OR p9

p9                  → p8 | p9 AND p8

p8                  → p7 | NOT p7

p7                  → p6 | p6 relation p6

p6                  → p5 | - p5 | p6 + p5 | p6 - p5

p5                  → p4 | p5 * p4 | p5 / p4 | p5 MOD p4

p4                  → p3 | p4 ↑ p3

p3                  → p2 | .p3 | @p3 | \ p3

p2                  → p1 | p1 <pointerparameters>

p1                  → literal |
```

```
                    name |
                    name [elist] |
                    p1 (elist) |
                    p 1 ( ) |
                    block |
                    (compoundexpression)
relation            → EQL | NEQ | LSS | LEQ | GTR | GEQ
pointerparameters   → position, size modification
modification        → | , index | , index, indirect
position            → | e
size                → | e
index               → | e
indirect            → | e
literal             → number | quotedstring
number              → decimal | octal
decimal             → digit | decimal digit
octal               → # oit | octal oit
digit               → 0 | 1 | 2 |...| 9
oit                 → 0 | 1 | 2 |...| 7
name                → letter | name letter | name digit
letter              → A | B | C | ... | Z | a | b | c | ... | z
quotedstring        → leftadjusted string | rightadjusted string
leftadjustedstring  → 'string'
rightadjustedstring → "string"
macrocall           → name (stringlist)
stringlist          → string | stringlist, string
```

The following list contains all the names reserved in the language:

AND	EXTERNAL	OF
AT	FORWARD	OR
BEGIN	FROM	OWN
BIND	FUNCTION	REGISTER
BY	GEQ	RETURN
CASE	GLOBAL	ROUTINE
CREATE	GTR	SELECT
DECR	IF	SET
ELSE	INCR	
END	LEQ	STRUCTURE
EQL	LOCAL	TES
EQV	LSS	TESN
EXCHJ	MACRO	THEN
EXIT	MAP	TO
EXITBLOCK	MOD	UNTIL
EXITCOMPOUND	NEQ	WHEN
EXITCOND	NOT	WHILE
EXITLOOP	NSET	XOR

APPENDIX B: INPUT-OUTPUT CODES*

The table beginning on the next page lists the complete teletype code. The lower case character set (codes 140-176) is not available on the Model 35, but giving one of these codes causes the teletype to print the corresponding upper case character. Other differences between the 35 and 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the console teletype, and the definitions bear no necessary relation to the use of the codes in conjunction with the PDP-10 software.

The line printer has the same codes and characters as the teletype. The 64-character printer has the figure and upper case sets, codes 040-137 (again, giving a lower case code prints the upper case character). The "96"-character printer has these plus the lower case set, codes 040-176. The latter printer actually has only ninety-five characters unless a special character is "hidden" under the delete code, 177. A hidden character is printed by sending its code prefixed by the delete code. Hence a character hidden under DEL is printed by sending the printer two 177s in a row.

Besides printing characters, the line printer responds to ten control characters, HT, CR, LF, VT, FF, DLE and DC1-4. The 128-character printer uses the entire set of 7-bit codes for printable characters, with characters hidden under the ten control characters that affect the printer and also under null and delete. In all cases, prefixing DEL causes the hidden character to be printed. The extra thirty-three characters that complete the set are ordered special for each installation.

The first page of the table of card codes [*pages*] lists the column punch required to represent any character in the two DEC codes. The octal codes listed are those used by the PDP-10 software. In other words, when reading cards, the Monitor translates the column punch into the octal code shown; when punching cards, it produces the listed column punch when given the corresponding code. The remaining pages of the table show the relationship between the DEC card codes and several IBM card punches. Each of the column punches is produced by a single key on any punch for which a character is listed, the character being that which is printed at the top of the card.

---

*This appendix reproduced with the permission of Digital Equipment Corporation from the PDP-10 Reference Handbook.

TELETYPE CODE

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed. Repeats on Model 37. Control shift P on Model 35.
1	001	SOH	Start of heading; also SOM, start of message. Control A.
1	002	STX	Start of text; also EOA, end of address. Control B.
0	003	ETX	End of text; also EOM, end of message. Control C.
1	004	EOT	End of transmission (END); shuts off TWX machines. Control D.
0	005	ENQ	Enquiry (ENQRY); also WRU, "Who are you?" Triggers identification ("Here is . . . ") at remote station if so equipped. Control E.
0	006	ACK	Acknowledge; also RU, "Are you . . . ?" Control F.
1	007	BEL	Rings the bell. Control G.
1	010	BS	Backspace; also FEO, format effector. Backspaces some machines. Repeats on Model 37. Control H on Model 35.
0	011	HT	Horizontal tab. Control I on Model 35.
0	012	LF	Line feed or line space (NEW LINE); advances paper to next line. Repeats on Model 37. Duplicated by control J on Model 35.
1	013	VT	Vertical tab (VTAB) Control K on Model 35.
0	014	FF	Form feed to top of next page (PAGE). Control L.
1	015	CR	Carriage return to beginning of line. Control M on Model 35.
1	016	SO	Shift out; changes ribbon color to red. Control N.
0	017	SI	Shift in; changes ribbon color to black. Control O.
1	020	DLE	Data link escape. Control P (DC0).
0	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	024	DC4	Device control 4, turns punch or auxiliary off. Control T (~~TAPE~~, AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, error. Control U.
1	026	SYN	Synchronous idle (SYNC). Control V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. Control W.
0	030	CAN	Cancel (CANCL). Control X.
1	031	EM	End of medium. Control Y.
1	032	SUB	Substitute. Control Z.
0	033	ESC	Escape, prefix. This code is generated by control shift K on Model 35, but the Monitor translates it to 175.
1	034	FS	File separator. Control shift L on Model 35.
0	035	GS	Group separator. Control shift M on Model 35.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	036	RS	Record separator. Control shift N on Model 35.
1	037	US	Unit separator. Control shift O on Model 35.
1	040	SP	Space.
0	041	!	
0	042	"	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	052	*	Repeats on Model 37.
0	053	+	
1	054	,	
0	055	-	Repeats on Model 37.
0	056	.	Repeats on Model 37.
1	057	/	
0	060	Ø	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	Repeats on Model 37.
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	Repeats on Model 37.
0	131	Y	
0	132	Z	
1	133	[	Shift K on Model 35.
0	134	\	Shift L on Model 35.
1	135	]	Shift M on Model 35.
1	136	↑	
0	137	←	Repeats on Model 37.
0	140		Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	Repeats on Model 37.
1	171	y	
1	172	z	
0	173	{	▲
1	174	|	
0	175	}	This code generated by ALT MODE on Model 35. ▲
0	176	~	This code generated by ESC key (if present) on Model 35, but the Monitor translates it to 175.
1	177	DEL	Delete, rub out. Repeats on Model 37.

Keys That Generate No Codes

REPT	Model 35 only: causes any other key that is struck to repeat continuously until REPT is released.
PAPER ADVANCE	Model 37 local line feed.
LOCAL RETURN	Model 37 local carriage return.
LOC LF	Model 35 local line feed.
LOC CR	Model 35 local carriage return.
INTERRUPT, BREAK	Opens the line (machine sends a continuous string of null characters).
PROCEED, BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 21-character message.

CARD CODES

▲ Character	PDP-10 ASCII	DEC 029	DEC 026	Character	PDP-10 ASCII	DEC 029	DEC 026
Space	040	*None*	*None*		100	8 4	8 4
!	041	11 8 2	12 8 7	A	101	12 1	12 1
"	042	8 7	0 8 5	B	102	12 2	12 2
#	043	8 3	0 8 6	C	103	12 3	12 3
$	044	11 8 3	11 8 3	D	104	12 4	12 4
%	045	0 8 4	0 8 7	E	105	12 5	12 5
&	046	12	11 8 7	F	106	12 6	12 6
'	047	8 5	8 6	G	107	12 7	12 7
(	050	12 8 5	0 8 4 ▲	H	110	12 8	12 8
)	051	11 8 5	12 8 4 ▲	I	111	12 9	12 9
*	052	11 8 4	11 8 4	J	112	11 1	11 1
+	053	12 8 6	12	K	113	11 2	11 2
,	054	0 8 3	0 8 3	L	114	11 3	11 3
−	055	11	11	M	115	11 4	11 4
.	056	12 8 3	12 8 3	N	116	11 5	11 5
/	057	0 1	0 1	O	117	11 6	11 6
0	060	0	0	P	120	11 7	11 7
1	061	1	1	Q	121	11 8	11 8
2	062	2	2	R	122	11 9	11 9
3	063	3	3	S	123	0 2	0 2
4	064	4	4	T	124	0 3	0 3
5	065	5	5	U	125	0 4	0 4
6	066	6	6	V	126	0 5	0 5
7	067	7	7	W	127	0 6	0 6
8	070	8	8	X	130	0 7	0 7
9	071	9	9	Y	131	0 8	0 8
:	072	8 2	11 8 2 *or* 11 0	Z	132	0 9	0 9
;	073	11 8 6	0 8 2	[	133	12 8 2	11 8 5
<	074	12 8 4	12 8 6	\	134	11 8 7	8 7
=	075	8 6	8 3	]	135	0 8 2	12 8 5
>	076	0 8 6	11 8 6	↑	136	12 8 7	8 5
?	077	0 8 7	12 8 2 *or* 12 0	←	137	0 8 5	8 2

Binary	7 9
Mode Switch	12 0 2 4 6 8
End of File	12 11 0 1

The octal codes given above are those generated by the Monitor from the column punches. The card reader interface actually supplies a direct binary equivalent of the column punch, as listed in the following two pages.

Column Punch	Character	Octal	Column Punch	Character	Octal
None	*Space*	0000	12 9	I	4001
0	0	1000	11 1	J	2400
1	1	0400	11 2	K	2200
2	2	0200	11 3	L	2100
3	3	0100	11 4	M	2040
4	4	0040	11 5	N	2020
5	5	0020	11 6	O	2010
6	6	0010	11 7	P	2004
7	7	0004	11 8	Q	2002
8	8	0002	11 9	R	2001
9	9	0001	0 1	/	1400
12 1	A	4400	0 2	S	1200
12 2	B	4200	0 3	T	1100
12 3	C	4100	0 4	U	1040
12 4	D	4040	0 5	V	1020
12 5	E	4020	0 6	W	1010
12 6	F	4010	0 7	X	1004
12 7	G	4004	0 8	Y	1002
12 8	H	4002	0 9	Z	1001

Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12	&	+	&	+	&	4000
11	–	–	–	–	–	2000
12 0				?		5000
11 0				:		3000
8 2				←	:	0202
8 3	#	=	#	=	#	0102
8 4	@	'	@	@	@	0042
8 5			'	↑	'	0022
8 6			=	'	=	0012
8 7			"	\	"	0006
12 8 2			¢	?	|	4202
12 8 3			.	.	.	4102
12 8 4	¤	)	<	)	<	4042
12 8 5			(	[	(	4022
12 8 6			+	<	+	4012

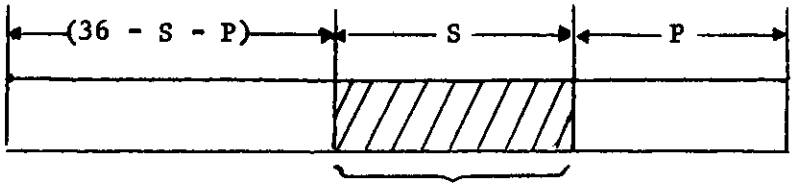
olumn Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12 8 7				!	↑	4006
11 8 2			!	:	!	2202
11 8 3	$	$	$	$	$	2102
11 8 4	*	*	*	*	*	2042
11 8 5			)	[	)	2022
11 8 6			;	>	;	2012
11 8 7			¬	&	\	2006
0 8 2			*See note*	;	]	1202
0 8 3	,	,	,	,	,	1102
0 8 4	%	(	%	(	%	1042
0 8 5			←	"	←	1022
0 8 6			>	#	>	1012
0 8 7			?	%	?	1006
12 11 0 1				*End of File*	*End of File*	7400
12 0 2 4 6 8				*Mode Switch*	*Mode Switch*	5252
7 9				*Binary*	*Binary*	*xx*05

NOTE: There is a single key for the 0 8 2 punch on the 029 but printing is suppressed.

The Monitor translates the octal code for the 12 0 punch in DEC 026 to 4202 (which corresponds to a , 12 8 2 punch), and the code for 11 0 to 2202 (11 8 2).

APPENDIX C: WORD FORMATS

<P,S> refers to a field S bits wide and P bits up from the right hand end of the word, thus:



referenced partial word

The format of a pointer is

P = <30,6>	Position
S = <24,6>	Size
I = <22,1>	Indirect address
X = <18,4>	IndeX
Y = <0,18>	

The format of an (non I/O) instruction is

F = <27,9>	Function code
A = <23,4>	Accumulator
I,X,Y as above	

The format of an integer number is

SIGN = <35,1>
MAGNITUDE = <0,35>

The format of a floating point number is

SIGN = <35,1>
EXPONENT = <27,8>
MANTISSA = <0,27>

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pennsylvania 15213

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

BLISS REFERENCE MANUAL

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Scientific Interim

5. AUTHOR(S) *(First name, middle initial, last name)*

W. A. Wulf, D. Russell, A. N. Habermann, C. Geschke,
J. Aperson and D. Wile

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
January 15, 1970	64	

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F44620-67-C-0058	
b. PROJECT NO. 9718	
c. 6154501R	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d. 681304	

10. DISTRIBUTION STATEMENT

1. This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH, OTHER	Air Force Office of Scientific Research (SRMA) 1400 Wilson Boulevard Arlington, Virginia 22209

13. ABSTRACT

This document describes the BLISS implementation language as written for the PDP-10. BLISS is a language specifically designed for use as a tool in implementing large software programs. Special attention is given in the language design to the requirements of the systems programming task, such as: space and time efficiency, the representation of data structures, the lack of run-time support facilities, flexible control structures, modularization, and parameterization of programs.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT