DECUS
PROGRAM LIBRARY

DECUS NO.	10-118 PART II
TITLE	A COLLECTION OF READINGS ON THE SUBJECT OF BLISS-1Ø
AUTHOR	Submitted by: M. G. Manugian
COMPANY	Digital Equipment Corporation Maynard, Massachusetts
DATE	December 1, 1971
SOURCE LANGUAGE	

ATTENTION

This is a USER program. Other than requiring that it conform to submittal and review standards, no quality control has been imposed upon this program by DECUS.

The DECUS Program Library is a clearing house only; it does not generate or test programs. No warranty, express or implied, is made by the contributor, Digital Equipment Computer Users Society or Digital Equipment Corporation as to the accuracy or functioning of the program or related material, and no responsibility is assumed by these parties in connection therewith.

A Collection of Readings on the Subject of

BLISS - 10

Printed December 1971
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

INTRODUCTION

The documents in this collection have been gathered together and reprinted in order to provide information pertaining to BLISS-10 not found in the BLISS-10 Reference Manual (DECUS 10-118, PDM 001-326-002-01). These documents serve three primary purposes. They provide a general description of the language and explain some of the basic, rather unique features of BLISS-10. They provide the background for a number of critical design choices in the language. Finally, they include examples and descriptions of some of the support software written for BLISS-10 as an aid to using the language.

The material presented in this document is for information purposes only. Digital Equipment Corporation makes no commitment to support any of the software as described herein.

Our thanks go to Professor William A. Wulf, Professor D. Russell, and Professor A. N. Habermann; also C. Geschke, J. Apperson, D. Wile and others at Carnegie-Mellon University through whose efforts the BLISS language was specified and implemented.

ACKNOWLEDGMENTS

Wulf, W. A. et. al., "BLISS - A Language for Systems Programming", CACM, December 1971, Copyright © 1971, Association for Computing Machinery, Inc. (Reprinting privileges were granted by permission of the Association for Computing Machinery.)

Wile, D. S. and C. M. Geschke, "Efficient Data Accessing in the Programming Language BLISS", SIGPLAN Symposium on Data Structures Proceedings, February 1971.

ABSTRACTS

Wulf, W. A. et. al., "BLISS - A Language for Systems Programming"

This paper discusses the design considerations in constructing a language especially suited for use in writing production software systems, e.g., compilers, loaders, and operating systems. BLISS, a language implemented at Carnegie-Mellon University for use in implementing software for the PDP-10, is described to illustrate the result of these considerations. Some comments are made on early experiences using BLISS for implementing various types of systems.

Geschke, C. et. al., "BLISS Examples"

This section contains a set of examples which illustrate the use of Bliss. Each example is intended to be fairly complete and self contained, and to illustrate one or more features of the language.

Wulf, W. A., "Programming Without the GOTO"

It has been proposed by Dijkstra and others that the use of the GOTO statement is a major contributing factor in programs which are difficult to understand and debug. This suggestion has met with considerable skepticism in some circles since GOTO is a control primitive from which a programmer may synthesize other, more complex, control structures which may not be available in a given language. This paper analyzes the nature of control structures which cannot be easily synthesized from simple conditional and loop constructs. This analysis is then used as the basis for the control structures of a particular language, BLISS, which does not have a GOTO statement. The results of two years of experience programming in BLISS, and hence without GOTO's , are summarized.

Wulf, W. A., "Why the DOT?"

An explanation of the pointer and contents concepts in BLISS justifying the semantic meaning of the dot operator. The current meaning is compared to possible alternative interpretations.

Wile, D. A. and C. M. Geschke, "Efficient Data Accessing in the Programming Language BLISS"

The specification of data structure in higher-level languages is isolated from the related specifications of data allocation and data type. Structure specification is claimed to be the definition of the accessing (addressing) function for items having the structure. Conventional techniques for data structure isolation in higher-level languages are examined and are found to suffer from a lack of clarity and efficiency.

The means by which data structure accessors may be defined in BLISS, the specification of their association with named allocated storage, and their automatic invocation by reference to the named storage only, are discussed. An example is presented which illustrates their efficient implementation and their utility for separating the activities of data structure programming and algorithmic programming.

Wulf, W. A., "HELP.DOC"

DDT may be used to debug programs written in BLISS; however, the use of DDT alone requires a fairly detailed knowledge of the run-time stack and other run-time characteristics of BLISS programs and is not especially convenient. In particular, DDT cannot exploit any special information about the structure of the object program. A module called "HELP" has been written to augment the facilities of DDT. This module may be loaded (along with DDT) with any BLISS program -- although recompilation of HELP is necessary if the user is not using the standard BLISS system registers. HELP is written in BLISS and therefore the facilities described below may be called directly from the user's source program even though they are primarily intended for use from DDT.

Wulf, W. A., "HELP.BLI"

This is the BLISS-10 source listing for the debugging aid described in HELP.DOC.

Newcomer, J. M., "TIMER.DOC"

This is the reference document and user manual for a package written in BLISS-10 which gathers a number of timing statistics for programs written in BLISS-10.

CONTENTS

BLISS - A Language for Systems Programming

William A. Wulf

BLISS
A LANGUAGE FOR SYSTEMS PROGRAMMING

W. A. Wulf, D. B. Russell, A. N. Habermann
Carnegie-Mellon University*
Pittsburgh, Pa.

ABSTRACT

This paper discusses the design considerations in constructing a language especially suited for use in writing production software systems, e.g., compilers, loaders, operating systems, etc. Bliss, a language implemented at Carnegie-Mellon University for use in implementing software for the PDP-10, is described to illustrate the result of these considerations. Some comments are made on early experiences using Bliss for implementing various types of systems.

INTRODUCTION

In the fall of 1969 Carnegie-Mellon University acquired a PDP-10 from Digital Equipment Corporation to support a research project on computer networks. This research will involve the production of a substantial number of large systems programs of the type which have usually been written in assembly language. At an early stage of this design effort it was decided not to use assembly language, but rather some higher level language. This decision immediately leads to another question: which language? In turn this leads to a consideration of the characteristics, if any, which are unique to, or at least exaggerated in, the production and maintenance of systems programs. The product of these deliberations was a new language which we call Bliss.

We refer to Bliss as an "implementation language", IL, although we admit that the term is somewhat ambiguous since, presumably all computer languages are used to implement something. To us the phrase connotes a general-purpose, higher-level language in which the primary emphasis has been placed upon a specific application, namely the writing of large, production software systems for a specific machine. Special purpose languages, such as compiler-compilers, do not fall into this catagorization, nor do we necessarily assume that these languages need be machine-independent. We stress the word 'implementation' in our definition and have not used words such as 'design' and 'documentation'. We do not necessarily expect that an implementation language will be an appropriate vehicle for expressing the design of a large system nor for the exclusive documentation of that system. Concepts such as machine-independence, expressing the design and implementation in the same notation, self-documentation, and others, are clearly desirable goals and are criteria by which we evaluated various languages. However, they are not implicit in our definition of the term "implementation language". There are a few extant examples of languages which fit our definition: EPL (a PL/I derivative used on MULTICS[1]), B5500 Extended Algol (Burroughs Corporation[2]), PL/360[3], and BCPL[4].

*This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F-44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research.

The various arguments for and against the use of higher level languages to write systems software have been discussed at length. We do not intend to reproduce them here in detail except to note that the skeptics argue primarily on two grounds: efficiency, and an assertion that the systems programmer must not allow anything to get between himself and the machine. The advocates argue on the grounds of production speed (and cost), maintainability, redesign and modification, understandability and correctness. The report of the NATO Conference on Software Engineering held in Garmish (October, 1968)[5] contains several discussions on these points, and the reader is urged to read that report.

It is our opinion that program efficiency, except possibly for a very small number of very small code segments, is determined by overall program design and not by locally tricky, "bit-picking" coding practices.

Many, if not all, systems have experienced substantial performance improvements from redesign or restructuring resulting from understanding or insight after the system has been running for some time. This redesign is frequently done by someone other than the program's original author. This argues for good documentation - but also for understandability of the code itself. Understandability is a function of many things, not all of which are inherent in the language in which a program is written - a programmer's individual style for example. Nevertheless, the length of a program text and the structure imposed upon that text are important factors and argue strongly for the use of a higher level language.

Presuming the decision to use an implementation language, which one should one choose? An argument might be made for choosing one of the existing languages, say Fortran, PL/I, or APL, and possibly extending it in some way rather than adding to the tower of Babel by defining yet another new one. We have chosen to do the latter and some justification is required. The only valid rationale for creating a new language is that the existing ones are inappropriate to the task. What then are the special characteristics of systems programs which existing languages are inappropriate to express? (Later we shall discuss how these manifest themselves in Bliss.) The two special characteristics most

frequently mentioned are efficiency and access to all hardware features of the machine. We add several things to these; the resulting list forms the design objectives of Bliss.

Requirements of Systems Programs
- space/time economy
- access to all relevant hardware features
- object code should not depend upon elaborate run-time support

Characteristics of Systems Programming Practice
- control over the representation of data structures
- flexible range of control structures (notably including recursion, co-routines, and asynchronous processes)
- modularization of a system into separately compilable sub-modules
- parameterization, especially conditional compilation

Overall Good Language Design
- encourage program structuring for understandability
- encourage program structuring for debugging
- economy of concepts (involution), generality, flexibility,...
- utility as a design tool
- machine independence

Not all of the goals mentioned above are compatible in practice, nor is the order in the above list accidental. Those found early in the list we consider to be absolute requirements while those occurring later in the list may be thought of as criteria by which alternative designs are judged once the more demanding requirements are satisfied.

For example, efficiency, access to machine features and machine independence are conflicting goals. In fact the design of Bliss is not machine independent, although the underlying philosophy and much of the specific design are. The machine for which the language was being designed, the PDP-10, was ever present in the minds of the designers. The code to be generated for each proposed construct, or form of a construct, was considered before that construct was included in, or excluded from, the language. Thus the characteristics of the target machine pervade the language in both overt and subtle ways. This is not to say that Bliss could not be implemented for another machine, it could. It does say that Bliss is particularly well suited to implementation on the PDP-10 and that it could probably not be as efficiently implemented on another machine. We think of Bliss as a member (the only one at present) of a class of languages similar in philosophy and mirroring a similar concern for the important aspects of systems programming, but each suited to its own host machine.

As another example of the incompatibility of these goals, consider the requirement for minimal run-time support and the use of the implementation language as a design tool. In some sense a design tool should be at a higher level than the object being designed - that is, the tool should relieve the designer from concern whichever details the designer deems appropriate only for later consideration. Any language relieves its user from concern over certain details, even assembly language frees the coder from the need to make specific address assignments. Assembly language is not a good design tool precisely because the class of such facilities is finite and narrow, a higher level language is better because the class is larger and broader. There is a point, however, beyond which broadening the class of details which are handled automatically introduces substantial costs in run-time efficiency and requisite run-time support. The design of Bliss walks a very fine line between generality, efficiency, and minimal run-time support. At the time of this writing Bliss programs require run-time support to the extent of one subroutine consisting of ten instructions.

DESCRIPTION OF BLISS

Bliss may be characterized as an Algol-PL/I derivative in the sense that it has a similar expression format and operator hierarchy, a block structure with lexically and dynamically local variables, similar conditional and looping constructs, and (potentially) recursive procedures. As may be seen from the two simple examples shown below the general format of Bliss code is quite Algol-like; however, the similarity stops shortly beyond this glib comparison.

```
function factorial (n) =
   if .n leq 1 then 0 else .n*factorial (.n-1);

function QQsearch (K) =
   begin register R,Q,A,E;
   E ← R ← .K/.n; Q ← .K mod .n; A ← .const;
   do if .ST[.R] eql .K
      then return .R
      else (R ← .R + .A; A ← .A + .Q)
   until .R eql .E
   end;
```

The first of these examples is the familiar recursive definition of factorial. The second example is the "quadratic quotient" hash search described by J. Bell in the February, 1970 CACM.

We will now describe the major features of Bliss in terms of its major aspects: (1) the underlying storage, (2) control, (3) data structures, and finally mention some other miscellaneous features.

1. Storage

A Bliss program operates with and on a number of storage "segments". A storage segment consists of a fixed and finite number of "words", each of which is composed of a fixed and finite number of "bits" (36 for the PDP-10). Any contiguous set of bits within a word is called a "field". Any field may be "named", the value of a name is called a "pointer" to that field. In particular, an entire word is a field and may be named.

In practice a segment generally contains either program or data, and if the latter, it is generally integer numbers, floating point numbers, characters, or pointers to other data. To a Bliss program, however, a field merely contains a pattern of bits. Various operations may be applied to fields and bit patterns such as fetching a bit pattern (value) from a field, storing a bit pattern into a field, integer arithmetic, comparison, boolean operations, and so on. The interpretation placed upon a particular bit pattern and consequent transformation performed by an operator is an intrinsic property of that operator

and not of its operands. That is to say, there is no 'type' differentation as in Algol.

Segments are introduced into a Bliss program by declarations, for example:

```
global g;
own x,y [5], z;
local p [100];
register r1, r2 [3];
function f(a,b) = .a↑.b;
```

Each of these declarations introduces one or more segments and binds the identifiers mentioned (e.g., g, x, y, etc.) to the name of the first word of the associated segment. (The function declaration also initializes the segment named 'f' to the appropriate machine code.)

The segments introduced by these declarations contain one or more words, where the size may be specified (as in "local p[100]"), defaulted to one as in "global g;"), or defaulted to whatever length is necessary for initialization (as in the function declaration). Explicit size declaration (as in "local p[100]") are restricted to expressions whose value can be determined at compile time so that run-time storage management is not required. The identifiers introduced by a declaration are lexically local to the block in which the declaration is made (that is, they obey the usual Algol scope rules) with one exception - namely, "global" identifiers are made available to other, separately compiled modules. Segments created by own, global, and function declarations are created only once and are preserved for the duration of the execution of a program. Segments created by local and register declarations are created at the time of block entry and are preserved only for the duration of the execution of that block. Register segments differ from local segments only in that they are allocated from the machine's array of 16 general purpose (fast) registers. Re-entry of a block before it is exited (by recursive function calls, for example) behaves as in Algol, that is, local and register segments are dynamically local to each incarnation of the block.

It is important to notice from the discussion above that identifiers are bound to names by these declarations, and that the value of a name is a pointer. Thus the value of an instance of an identifier, say x, is not the value of the field named by x, but rather is a pointer to x. This interpretation requires a "contents of" operator for which the symbol "." has been chosen. (Which explains the occurrence of this character in the earlier examples. This will be discussed in much greater detail under the subject of data structures.) There are two additional declarations whose effect is to bind identifiers to names, but which do not create segments; examples are:

```
external  s;
bind      y2 = y+2, pa = p+.a;
```

An external declaration binds one or more identifiers to the names represented by the same name declared global in another, separately compiled module. The bind declaration binds one or more identifiers to the value of an expression at block entry time. This will be discussed in greater detail in the section on data structures.

2. Control

Bliss is an "expression language", that is, every executable construct, including those which manifest control, is an expression and computes a value. There are no statements in the sense of Algol or PL/I. Expressions may be concatenated with a ";" to form compound expressions, where the value of a compound expression is that of its last component expression. Thus ";" may be thought of as a dyadic operator whose value is simply that of its righthand operand. The grouping symbols "begin" and "end" or "(" and ")" may be used to embrace such a compound expression and convert it into a simple expression. A block is merely a special case of either of these constructions which happens to contain declarations, thus the value of a block is defined to be the value of its constituent compound expression.

The assignment operator, "←", is a dyadic operator whose left operand is interpreted as a pointer and whose right operand is an uninterpreted bit pattern. The right operand is stored into the field named by the left operand, the value of the expression is that of its right operand. Recalling the interpretation of identifiers and the "." operator, the expression

```
x←.x+1
```

causes the value of the field named by x to be incremented by one. The value of the entire assignment expression is that of the incremented value. The compound expression

```
(y ← x; z ← ..y+1)
```

causes a pointer to x to be stored into y, then computes the value of the field named by x (accessed indirectly through y) plus one and stores this value in z; this value is also that of the compound expression.

There is the usual complement of arithmetic, logical, and relational operators. Logical operators operate on all bits of a word; relational operators yield a value 1 if the relation is satisfied and a value of 0 otherwise.

We will describe six forms of control expressions: conditional, looping, case-select, function call, co-routine call, and escape. For this discussion it will be convenient to use the symbol ϵ, possibly subscripted, to represent an arbitrary expression.

The conditional expression is of the form

$$\underline{\text{if}}\ \epsilon_1\ \underline{\text{then}}\ \epsilon_2\ \underline{\text{else}}\ \epsilon_3$$

and is defined to have the value ϵ_2 just in the case that the rightmost bit of ϵ_1 is a 1 and has the value of ϵ_3 otherwise. The abbreviated form "$\underline{\text{if}}\ \epsilon_1\ \underline{\text{then}}\ \epsilon_2$" is considered to be identical to "$\underline{\text{if}}\ \epsilon_1\ \underline{\text{then}}\ \epsilon_2\ \underline{\text{else}}$ 0".

There are four basic forms of looping expressions:

$$\underline{\text{while}}\ \epsilon_1\ \underline{\text{do}}\ \epsilon$$
$$\underline{\text{do}}\ \epsilon\ \underline{\text{while}}\ \epsilon_1$$

incr <name> from ϵ_1 to ϵ_2 by ϵ_3 do ϵ
decr <name> from ϵ_1 to ϵ_2 by ϵ_3 do ϵ

Each form of looping expression implies repeated execution (possibly zero times) of the expression denoted ϵ until a specific condition is satisfied. In the first form the expression (while...do) ϵ is repeated so long as the rightmost bit of ϵ_1 remains 1. The second form is similar to the first except that ϵ is evaluated before ϵ_1 thus guaranteeing at least one execution of ϵ. The last two forms are similar to the familiar "step...until" construct of Algol, except (1) the control variable is local to ϵ, (2) ϵ_1, ϵ_2, and ϵ_3 are computed only once (before entry to the loop), and (3) the direction of the step is explicitly indicated (increment or decrement). Except for the possibility of an escape expression within ϵ (see below) the value of a loop expression is uniformly taken to be -1.

We shall treat somewhat simplified versions of the case and select expressions here, these forms are:

case e of set ϵ_0; ϵ_1; ...; ϵ_{n-1}; ϵ_n tes
select e of nset ϵ_0:ϵ_1; ϵ_2:ϵ_3; ...; ϵ_{2n}:ϵ_{2n+1} tesn

The value of a case expression is ϵ_e, that is, the expression e is evaluated and this value is used to select one of the expressions ϵ_i $(0 \le i \le n)$ whose value, in turn, becomes the value of the entire case expression. The select expression is somewhat similar to the case expression with the distinction that the value of e is not restricted to the range $0 \le e \le n$. Execution of the select proceeds as follows: (1) the value of e is computed, (2) the value of the expressions ϵ_{2i} $(0 \le i \le n)$ are evaluated, (3) for each i such that $e = \epsilon_{2i}$ the expression ϵ_{2i+1} is evaluated. Thus, in the event that more than one value of i exists such that $e = \epsilon_{2i}$, each of these expressions is evaluated; in this case the final value of the select expression is undefined.

A function call expression has the form

$$\epsilon(\epsilon_1, \epsilon_2, \ldots \epsilon_n)$$

This expression causes activation of the segment named by ϵ as a subprogram with an initialization of the formal parameters named in the declaration of the function to the values of the actual parameters $\epsilon_1, \ldots, \epsilon_n$. Only call-by-value parameters are allowed; however, call-by-reference is available since names, pointer values, may be passed. The value of a function call is that resulting from execution of the body of the function. Thus, for example, the value of the following block is 3628800.

```
begin
function factorial(n) =
   if .n leq 1 then 1 else .n*factorial(.n-1);
factorial(10)
end
```

Note that a function call need not explicitly name a function by its associated identifier; all that is required is that ϵ evaluate to the name of a segment. Thus expressions such as the following are valid and useful.

(case.x of set P1;P2;P3 tes)(.z)

Also note that the occurrence of a parameter list enclosed in brackets triggers a function call. An identifier by itself merely denotes a pointer to the named segment; thus in the example above P1, P2, and P3 are the names of functions and thus the value of the case statement is the name of one of these functions (not the result of executing it). Function calls with no parameters are written "ϵ( )".

The body of any function may be activated as a co-routine and/or asynchronous process. An arbitrary number of distinct incarnation of a single body are allowed. In order to permit any of several realizations of co-routine mechanisms only two primitive operations are provided.

create $\epsilon(\epsilon^1, \epsilon^2, \ldots, \epsilon^n)$ at ϵ_2 length ϵ_3 then ϵ_4
exchj(ϵ_5,ϵ_6)

The effect of the create expression is to create an independent context (that is, a stack) for the function named by ϵ with parameters $\epsilon^1, \ldots, \epsilon^n$. The stack is set up beginning at the word named by ϵ_2 and is of size ϵ_3 words (to provide overflow protection). The activation record for the newly created co-routine is set to the head of the function named by ϵ. The value of the create expression is a "process name" for the new co-routine. Control then passes on to the expression following the 'create' - in particular the expression ϵ_4 is not executed at this time and the body of ϵ is not activated. When two or more such contexts have been established, control may be passed from the currently executing one to any other by executing an exchange jump, exchj, expression. An expression "exchj(ϵ_5,ϵ_6)" will cause control to pass to the co-routine named by ϵ_5 (the value of an earlier create expression). The value ϵ_6 becomes the value of the exchj operation which last cause control to pass out of the co-routine named by ϵ_5.

The familiar "goto...label" form of control has not been included in Bliss. There are two reasons for this: (1) unrestricted goto's require considerable run-time support due to the possibility of jumping out of functions and/or blocks, and (2) the authors feel strongly that the general goto, because of the implied violation of program structure, is a major contributor to making programs difficult to understand, modify and debug. There are "good" and "bad" ways to use a goto and there are restrictions which could be imposed which eliminate the need for run-time support. Consideration of the nature of "good" ways and the restrictions necessary to eliminate run-time overhead led us to eliminate the goto altogether, and to the inclusion of conditional, looping, and case-select expressions. These alone, however, are not sufficiently general, or convenient, and consequently the 'escape' expressions were introduced. There are six forms of escape expressions:

EXITBLOCK	ϵ	EXITCOND	ϵ
EXITCOMPOUND	ϵ	EXIT	ϵ
EXITLOOP	ϵ	RETURN	ϵ

Each form of escape expression causes control to exit from a specified control environment (a block, a loop, or a conditional expression, for example) and defines a value (ϵ) for that control expression (EXIT exits from any form of control expression, RETURN exits from a function).

Consider a linked list of two word cells, the first of which contains a link (pointer) to the next

cell (the last cell has link=0) and the second of which contains data. The following expression has a value which is the pointer to the first negative data item, or a value of -1 if no such item is found. The address of the head of the list is contained in a field called 'head'.

```
(register t; t ← head; while (t ← .t) neq 0 do if
 .(.t+1) lss 0 then break .t);
```

Note that the initialization of t, i.e., 't ← head', sets the value of 't' to a pointer to 'head', not the contents of 'head'.

3. Data Structures

One of the outstanding characteristics of systems programs is their concern with the wide variety of data structures and schemes for representing these structures. Observation of what systems programmers do reveals that a very large fraction (nearly 50% in our experience) of their design effort is spent in designing representations for efficiently encoding the information they will process. It is frequently the case that the most difficult task in making a modification to an existing program is that of representing the additional new information required (e.g., the infamous "find another bit" problem). Consequently the issue of representation was one of the central design considerations in Bliss.

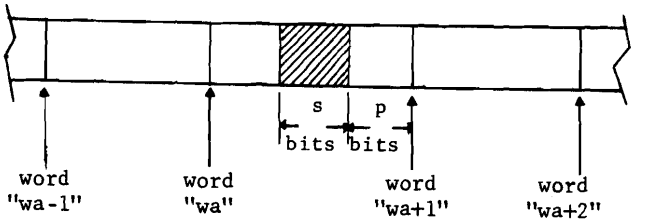
Two principles were followed in the design of the data structure facility of Bliss:

- the user must be able to specify the accessing algorithm for elements of a structure,
- the representational specification and the specification of algorithms which operate on the represented information must be separated in such a way that either can be modified without affecting the other.

The first principle follows simply from the fact that non-algorithmic specifications are inadequate to express certain important representational schemes. By a non-algorithmic specification we mean one which statically specifies the layout of a structure in terms of primitive structures (words, fields, etc.), other defined structures, and (possibly) pointers. By an algorithmic specification we mean one which, given a set of parameters (indices) computes a pointer to the appropriate structure element. Algorithmic specifications have the advantage of generality, but some disadvantage of verbosity for simple structures. This latter type of specification will be amply illustrated below.

In order to achieve a language in terms of which it is possible to write large systems that may be easily modified, it is imperative that the specifications of the representation of a data structure be separated from the specification of algorithms which manipulate data in that structure. This principle is severely violated in assembly languages where, typically, the code to access an element of a structure, for example, simply a contiguous field of bits within a word, is coded "in line" at the point where the element is needed. A comparatively trivial change which alters the size or position of the field and may require locating and modifying all references to the field. This simple problem could be solved by following good coding practice and, perhaps, by the use of macros; not all changes are of such a trivial nature, however.

The concept of a "pointer" to a field (of bits within a word) was mentioned earlier. Actually in Bliss a pointer is a five-tuple consisting of: (1) a word address, (2) a field position, (3) a field size, (4) an (index) register name, and (5) an "indirect address" bit. These five quantities are encoded in a single word and as such are a manipulatable item in the language (a prerequisite of algorithmic representational specification). For simplicity we shall discuss only the first three of these quantities; the reader is referred to the Bliss reference manual[6] for more detail. The "word address", wa, field of a pointer designates the physical machine address of the word; the 'position', p, and 'size', s, designate a field within a word in terms of the number of bits to the right of and within the field.



The notation used in Bliss to specify a pointer (taking only the simple wa,p,s case) is "wa<p,s>".

Assume that the declaration

```
own x[100]
```

has been made. The identifier x is bound by this declaration to a pointer to the 36 bit field which is the first word of this 100 word segment. That is, the word address of the pointer "x" is that of the location allocated to the segment and the position and size fields have values of zero and thirty-six respectively. If we denote the address of the segment by α_x, then an occurrence of "x" in a Bliss program is identical to an occurrence of "α_x<0,36>". If ϵ_0 - ϵ_2 are expressions, then the syntactic form

$$\epsilon_0<\epsilon_1,\epsilon_2>$$

is by definition a pointer whose word address is the value of ϵ_0 (modulo 2^{18}) and whose position and size specifications are the values of ϵ_1 and ϵ_2 (modulo 2^6) respectively. Thus "X<3,4>" is a pointer to a four bit field three bits from the right end of a word named X. The word address, position, and size information are encoded within a pointer in such a way that adding small integers ($<2^{18}$) to a pointer increments the word address only. Thus "X+1" is a pointer to the word following X.

The definition of a class of structures, that is, of an accessing algorithm to be associated with certain specific data structures, is made by a declaration of the form:

```
structure <name>[<formal parameter list>] = ϵ
```

Particular names may then be associated with an accessing algorithm by another declaration

```
map <name>:<name list>
```

Consider the following example:

```
begin
  structure ary2[i,j] = (.ary2+(.i-1)*10+(.j-1));
  own x[100],y[100],z[100];
  map ary2: x,y,z;
  :
  x[.a,.b] ← .y[.b,.a];
  :
end;
```

In this example we introduce a very simple structure, ary2, for two dimensional (10x10) arrays, declare three segments with names 'x', 'y', and 'z' bound to them, and associate the structure class ary2 with these names. The syntactic forms "$x[\epsilon_1,\epsilon_2]$" and "$y[\epsilon_3,\epsilon_4]$" are valid within this block and denote evaluation of the accessing algorithm defined by the structure declaration (with an appropriate substitution of actual for formal parameters). Within the expression defining a structure class, the name of the structure class, ary2 in this case, denotes the name of the "zeroth" formal parameter - and is replaced by the name preceding the "[" at the call site. Thus, ".ary2" denotes the value of the name of the particular segment being referenced. In the example 'x[.a,.b]' is equivalent to:

```
(x+(.a-1)*10+(.b-1))
```

The value of this expression is a pointer to the designated element of the segment.

In the following example the structure facility and bind declaration have been used to efficiently encode a matrix product

$$(z_{i,j} = \sum_{k=1}^{10} x_{ik} y_{kj}).$$

In the inner block the names 'xr' and 'yc' are bound to pointers to the base of a specified row of x and column of y respectively. These identifiers are then associated with structure classes which allow one-dimensional access.

```
begin
  structure ary2[i,j] = (.ary2+(.i-1)*10+(.j-1)),
            row[i] = (.row+.i-1),
            col[j] = (.col+(.j-1)*10);
  own       x[100],y[100],z[100];
  map       ary2: x,y,z;
  :
  incr i from 1 to 10 do
    begin bind xr=x[.i,1],zr=z[.i,1];map row:xr,zr;
    incr j from 1 to 10 do
      begin
        register t; bind yc=y[1,.j];map col:yc;
        t ← 0;
        incr k from 1 to do t ← .t+.xr[.k]*.yc[.k];
        z[.j] ← .t;
      end;
    end;
  :
end
```

Suppose now that one wishes to alter the representation of the structure 'ary2', and access to the array is to be made through an Ilife vector (or, "dope" vector) to define the relative base of each row. The major change required is to replace the current structure declaration for "ary2" by

```
own il[10]; map row: il;
structure ary2[i,j] = (.ary2+.il[.i-1]+.j-1);
```

With this representation, the use of a special accessing algorithm (structure) for accessing columns becomes

```
structure col[j] = (.col + .il[.j-1]);
```

As can be seen, these fairly simple changes to the program completely changes its representation of the data. No changes to the processing algorithm are required.

4. Miscellaneous Features

Finally, we shall now describe two features of the language which are important to the goal of parameterization of programs. The first is simply that constant expressions are evaluated at compile time. This is a common feature of compilers and not particularly exciting by itself. Note, however, that since the value '1' is interpreted as true, and '0' as false, expressions such as

```
if 1 and 1 or 0 then ... else ... ;
```

are constant in that only the then part will be executed. The compiler notes this and does not emit the code for testing the condition or evaluating the else part. Similarly, only the third expression of the following case expression will be evaluated at execution time, and consequently the compiler only generates code for that expression.

```
case -1+2+1 of set ϵ0; ϵ1; ϵ2; ... tes;
```

The second feature is a fairly elementary string replacement macro capability. A macro name and its associated text are introduced by a declaration of the form:

```
macro    ntapes = 3$,
         ndrums = 5$,
         loop(i,n) = incr i from 1 to n do $;
```

This particular declaration defines three macro names ('ntapes', 'ndrums', and 'loop') and defines a text string which is to replace the macro name (and its parameters, if any) where it (they) is (are) mentioned in the scope of the declaration. The end of a text string is delimited by '$', and may mention formal parameter names - these are replaced by actual parameter strings used at the call site.

One may combine these two features to parameterize a system. Consider the following skeletal code:

```
begin
  macro      ntapes = 3$,
             ndrums = 5$,
             descsize = 2$,
             cloop(i,n) = if n gtr 0 then incr i
                          from 1 to n do $;
  own        devicedesc [ntapes*ndrums*descsize];
  structure  devary [i,j] = (.devary + (.i-1)*desc-
                            size + .j);
  map        devary: devicedesc;
```

The declarations above define a table of device descriptions for magnetic tapes and drums. The number of entries for tapes and drums, and the number of words per description entry are controlled by the macro definitions 'ntapes', 'ndrums', and 'descsize'. Suppose the number and size of fields within the device description for tapes and drums are different. The following structure and bind declarations allow one to access these fields conveniently:

```
structure tapeary [i,j] =
   case (.j-1) of
      set
      (.tapeary + (.i-1)*descsize)<0,36>;
      (.tapeary + (.i-1)*descsize)<18,18>;
      (.tapeary + (.i-1)*descsize)<18,18>;
      tes;
structure drumary [i,j] =
   case (.j-1) of
      set
      (.drumary + (.i-1)*descsize)<0,36>;
      (.drumary + (.i-1)*descsize+1)<18,18>;
      (.drumary + (.i-1)*descsize+1)<17,1>;
      (.drumary + (.i-1)*descsize+1)<16,1>;
      (.drumary + (.i-1)*descsize+1)<0,16>;
      tes;
bind     tapedesc = devicedesc [0,0],
         drumdesc = devicedesc [ntapes,0];
map      tapeary: tapedesc, drumary: drumdesc;
```

These declarations make it feasible for the programmer to refer to 'tapedesc [.i,2]', for example, as the second field of the description of the ith tape without regard to the size or location of that field. The following code uses the constant expression evaluation feature to selectively include only relevant code.

```
global function initialize =
  begin
    cloop(i,ntapes)
      begin
         % code to initialize tape description goes
           here%
      end;
    cloop(i,ndrums)
      begin
        % code to initialize drum descriptions goes
          here %
      end;
    % other initialization code goes here %
  end;
if ntapes gtr 0 then
  begin
    global function tapehandler =
      begin
        % code for body of tape device handler %
      end;
    global function tapeopen =
      begin
        % code for special file-open actions on
          magnetic tape %
      end;
    % other specialized tape functions declared
      here %
end
```

Since the body of an "if ϵ_1 then ϵ_2" expression is not compiled in the case that the ϵ_1 is a constant, and false, the global functions 'tapehandler', etc., are not compiled unless 'ntapes' is greater than zero. One can imagine more complex expressions, such as 'if (ndrums gtr 0) or (ndisks gtr 0) then', controlling the inclusion of, for example, file-directory handling code.

EVALUATION AND CONCLUSIONS

As of this writing the Bliss compiler is in its final stages of completion, and consequently experience using the language is somewhat limited. To date only one major project has been undertaken in Bliss, namely the compiler itself. The language has evolved as a consequence of this experience, and we expect it will evolve further as it is used.

In spite of the relative lack of experience in using the language, it would be very nice to have some objective measures of the language - measures of such things as efficiency, appropriateness (to the systems programming problem), readability, consistency, etc. Such measures are, of course, very difficult to define objectively. However, we have attempted to supply some data from which the user may draw his own conclusions. One of these data points indicates the quality of code produced by the Bliss compiler - and is therefore an indirect measure of the suitability of the language for one system's programming problem. The second bit of data is an annoted table comparing features of some implementation languages.

The measure chosen for code quality of the Bliss compiler is simply that of code size. Three sections of the compiler were chosen as a basis for comparison in an attempt to factor out those things which (1) are intrinsic to the structure of the language, (2) are a function of the current optimization strategies of the compiler (which can always be improved), and (3) are a function of a particular programmer's "style". The sections are named IO, LEXAN, and SYNTAX and are respectively the i/o interface, lexical analyzer (symbol table routines, etc.), and syntax analyzer. Of these, IO was originally written in "clever" assembly code and later translated into Bliss, while LEXAN and SYNTAX were originally written in Bliss and then translated by hand into assembly code. The translation of LEXAN was done in such a way as to mirror the functional structure of the original Bliss code at the subroutine level but internally was coded for maximal efficiency. SYNTAX, on the other hand, was translated with the aid of a number of general purpose macros and mirrors exactly the structure of the original Bliss text. The results are as follows:

	approximate size	relative size of compiled version
IO	50	40% larger
LEXAN	1300	7% larger
SYNTAX	2300	20% smaller

From this small sample one can draw some tentative conclusions:

1. IO is something like a worst case. It is small (which tends to exaggerate the overhead for recursion, etc.) and it was originally written in assembly code. The penality in such a case appears to be on the order of 50%.

2. Since the hand coding of LEXAN obeys the subroutine calling conventions of compiled Bliss programs, but is otherwise coded

fairly tightly - the penality for the current optimization techniques appears to be on the order of 10%.

3. The compiler does considerably better than macro extension of assembly code.

Table I and its associated notes compare certain features of implementation languages as described by the most recent documentation available to these authors, and speaks for itself. Neither the list of features nor the list of languages is exhaustive; both reflect the prejudcies of the authors. Numbers in the lower right corner of entries refer to the notes following the table.

The comparisons of code size and language features given above hopefully provide some insight into the use of Bliss as an implementation tool; unfortunately, they do not give absolute measures of its utility. In particular there seems to be no way at present to measure the benefits of maintainability and modifyability - and these are, in the opinion of the authors, its major advantage.

NOTES ON TABLE I

1. Of course no language is explicitly designed to produce large, slow programs. The entries in this row reflect the extent to which efficiency was a prime goal and the extent to which concessions were made.

2. Bliss and Espol have limited macro facilities when compared to most macro assemblers, namely, simple string replacement (with parameters). PL/I has extensive macro facilities, but these are not described as part of EPL.

3. All of the languages listed either have the ability to embed assembly code or to call machine language subroutines. The entry relates principally to the former facility.

4. The entries are coded as follows:

   M machine data types
   C conceptual data types
   op type interpretation is derived from operator
   V type interpretation is derived from variables
   D type interpretation is derived from data

5. 'na' denotes 'not applicable'.

6. Fortran, Espol and EPL provide no control over the representation of data structures. Macro, BCPL, SAL, and PL/360 provide such control; however the access to elements of structures must be programmed "in-line".

7. Macro, SAL, and PL/360 permit recursions in the sense that the programmer may choose to explicitly code a recursive calling sequence.

8. The following code are used to denote various parameter passing options.

   V call-by-value
   N call-by-name
   R call-by-reference

9. The following codes are used to denote various control statement forms:

   I_0 Fortran if-statement
   I_1 Algol-like "if-then-else"
   D Do-statement
   F Algol-like for-statement
   C case statement
   SF simple-for (corresponds to Algol step-until case)
   W while-statement
   G goto
   SO BCPL "switchon", similar to Bliss "select"

ACKNOWLEDGMENTS

We would like to express our deep gratitude to Messrs. Geschke, Wile, and Apperson (graduate students at CMU) each of whom has made valuable contributions to both the design and implementation of the language.

REFERENCES

1. EPL Reference Manual, Project MAC, April 1966.

2. "Burroughs B5500 Extended Algol Reference Manual", Burroughs Corporation, Detroit, Mich.

3. Wirth, N., "PL/360, A Programming Language for the 360 Computers", JACM, 15, 1, Jan. 1968, p. 37.

4. Richards, M., "BCPL: A tool for compiler writing and system programming", SJCC, 1969, p. 557.

5. Naur, P. and B. Randell (Ed.), "Software Engineering", Scientific Affairs Division, NATO, Brussels, Belgium. (Held in January 1969 in Garmish.)

6. "Bliss Reference Manual", Computer Science Department Report, Carnegie-Mellon University, Pittsburgh, Pa., Jan. 15, 1970.

7. "PDP-10 Reference Handbook", Digital Equipment Corporation, Maynard, Mass., 1970.

8. Lang, Charles A., "SAL - Systems Assembly Language", SJCC, 1969, p. 543.

TABLE I

COMPARISON OF IMPLEMENTATION LANGUAGES

GOAL FEATURE	BLISS	MACRO-10	FORTRAN	B5500 ESPOL	EPL (PL/I)	BCPL	SAL	PL/360
space/time economy [1]	goal	goal	goal [2]	goal	no	no	goal	goal
machine independence	no	no	yes	no	yes	yes	no	no
Macro	yes [2]	yes [2]	no	yes [2]	no	no	no	no
access to hardware [3]	yes	yes	no	yes	no	no	yes	yes
run time support required	no	no	yes	some	yes	some	no	no
data types [4]	M,op	M,op	C,V (real, integer)	C,(D,V) (real, integer, pointer)	C,V (real, integer, character, pointer)	M,op	C,op (real, integer)	C,V (real, integer)
automatic conversion of data types [5]	na	na	yes	yes	yes	na	na	no
data structures	user defined	user defined	arrays	arrays	hier-archical	vectors	user defined	user defined
control of representation of structures	yes	yes	no	no	no	yes	yes	yes
recursion	yes	yes [7]	no	yes	yes	yes	no [7]	yes [7]
co-routines	yes	yes	no	no	no	no	no	no
parameters [8]	V,R	na	R	V,N,R	V,R	V,R	V,R	V,R
conditional/ looping, etc. [9]	see text	na	I_0,D,G	I_1,F,C,G	I_1,D,G	I_1,SF,W, C,SO,G	I_1,SF,W,G	I_1,SF,W, C,G
References	[6]	[7]		[2]	[1]	[4]	[8]	[3]

BLISS Examples

C. M. Geschke *et. al.*

SECTION VI

BLISS EXAMPLES

This section contains a set of examples which illustrate the use of Bliss. Each example is intended to be fairly complete and self contained, and to illustrate one or more features of the language.

The authors would like to invite others to contribute further examples for inclusion in this section. New examples will be included if they clearly illustrate features and/or uses of the language which are not already adequately illustrated.

EXAMPLE 1: A TT-CALL I/O PACKAGE

Contributors: C. Geschke and W. Wulf

The following set of declarations defines a set of teletype input/output routines using the PDP-10 monitor TT-call mechanism. The set of functions is not complete, but adequate to illustrate the approach.

The declarations below provide the following functions:

INC	Input one character - wait for EOL before returning
OUTC	Output one character
OUTSA	Output ASCIZ-type string beginning at specified address
OUTS	Output ASCIZ-type string specified as the parameter
OUTM	Output multiple copies of a specified character
CR	Output carriage return
LF	Output line feed
NULL	Output null character
CRLF	Output carriage return and line-feed followed by 2 nulls
TAB	Output tab
OUTN	Output number in specified base and minimum number of digits
OUTD	Output decimal number with at least one digit
OUTO	Output octal number with at least one digit
OUTDR	Output decimal number with at least specified number of digits
OUTOR	Same as OUTDR except octal

```
MODULE TTIO(STACK)=BEGIN

MACHOP TTCALL=#51;

MACRO    INC= (REGISTER Q; TTCALL(4,Q); .Q)$,
         OUTC(Z)= (REGISTER Q; Q←(Z); TTCALL(1,Q))$,
         OUTSA(Z)= TTCALL(3,Z)$,
         OUTS(Z)= OUTSA(PLIT ASCIZ Z)$,
         OUTM(C,N)= DECR I FROM (N)-1 TO 0 DO OUTC(C)$,
         CR= OUTC(#15)$,  LF= OUTC(#12)$,  NULL= OUTC(0)$,
         CRLF= OUTS('?M?J?0?0')$,
         TAB= OUTC(#11)$;


ROUTINE OUTN(NUM,BASE,REQD)=
  BEGIN OWN N,B,RD,T;
    ROUTINE XN=
      BEGIN LOCAL R;
        IF .N EQL 0 THEN RETURN OUTM("0",.RD-.T);
        R←.N MOD .B; N←.N/.B; T←.T+1; XN();
        OUTC(.R+"0")
      END;

    IF .NUM LSS 0 THEN OUTC("-");
    B←.BASE; RD←.REQD; T←0; N←ABS(.NUM); XN()
  END;



MACRO    OUTD(Z)= OUTN(Z,10,1)$,
         OUTO(Z)= OUTN(Z,8,1)$,
         OUTDR(Z,N)= OUTN(Z,10,N)$,
         OUTOR(Z,N)= OUTN(Z,8,N)$;


!        THE PROGRAM BELOW PRINTS A TABLE OF INTEGERS, THEIR SQUARES, AND
!  THEIR CUBES:


    OWN N,C;

    CRLF; OUTS('INPUT AN INTEGER PLEASE ...');
    N←0; WHILE (C←INC) GTR "0" AND .C LSS "9" DO N←.N*10+(.C-"0");

    CRLF; OUTS('A TABLE OF THE SQUARES AND CUBES OF 1-'); OUTD(.N);

    CRLF; INCR I FROM 1 TO 3 DO (TAB; OUTS(' X↑'); OUTD(.I));

    CRLF; INCR I FROM 1 TO 3 DO (TAB; OUTM("-",5));

    INCR I FROM 1 TO .N DO
      BEGIN OWN X;
        X←.I; CRLF;
        DECR J FROM 2 TO 0 DO (TAB; OUTD(.X); X←.X*.I)
      END

END ELUDOM
```

Although the example is quite simple, there are several things about it which should be noted:

1. The use of a MACHOP declaration and embedded assembly code.
2. The use of macros to add a level of "syntactic sugar" and general cleanliness to the code.
3. The use of the escape character "?" in the CRLF macro to obtain control characters (e.g., carriage-return) in strings.
4. Parenthesization of macro parameters, as in OUTM, to insure proper hierarchy relations in the expansion.
5. The use of "DECR-TO-ZERO" in OUTM because it produces better code than "INCR-TO-EXPRESSION".
6. The use of own variables and the parameterless procedure XN in OUTN in order to avoid passing redundant parameters through the recursive levels of XN.
7. The fact that the local variable "R" is local to each recursive level of XN and hence its value is preserved at each level.

EXAMPLE 2: QUEUE MANAGEMENT MODEL

Contributors: C. Geschke and W. Wulf

This module contains routines to insert and delete items on doubly-linked queues. In addition it contains space management routines implementing the "Buddy System" (cf: Knuth: Vol. 1).

Buddy System

This is not intended to be a detailed description of the buddy system model of space management. We will simply give a brief description of this implementation of the scheme. The vector of allocatable space is called MEM. Space is allocated and deallocated from MEM by the routines GET and RELEASE, respectively. The basic unit of allocatable space is an item. Items are of size 2**ITEMSIZE where 0 < ITEMSIZE ≤ LOG2MEMSIZE. The first two words of an item are formatted:

ITEMSIZE	RLINK
<NOT-USED>	LLINK

Available items of size N are elements of a doubly linked list whose header is the two word cell SPACE[N]. The routines LINK and DELINK are called to enter and remove items from lists. The routine COLLAPSE is used to compactify two adjacent available items of size 2**N into an item of size 2**(N+1). The COLLAPSE routine iterates this process until no more compactification can take place.

Queue Model

In this model a queue is defined to be a doubly-linked list suspended from a header whose first three words are formatted as follows:

HEADERSIZE	RLINK
<NOT-USED>	LLINK
REMOVE	ENTER

The fields REMOVE and ENTER contain the addresses of the routines to be invoked when removing and entering items on the queue. To enter item X on queue Q, one simply makes the call ENQ(X,Q). ENQ then invokes the enter routine in Q's header which returns the address of the item in Q after which X is to be inserted. In a similar manner one removes the "next" item from queue Q by the call DEQ(Q). DEQ then invokes the remove routine in Q's header to return the address of the "next" item. The advantage of this scheme is that the queueing discipline is queue specific, and the same primitives (ENQ and DEQ) may be used independent of the discipline used for that queue. Examples of the enter and remove routines for LIFO, FIFO, and PRIORITY type queues appear at the end of this example module.

```
MODULE QMS(STACK)=

! BUDDY SYSTEM
!--------------

BEGIN

BIND MEMSIZE=1^12;

GLOBAL VECTOR MEM[MEMSIZE];

BIND LOG2MEMSIZE=35-FIRSTONE(MEMSIZE);

STRUCTURE ITEM[I,J,P,S]=
        CASE .I OF
          SET
            (.ITEM)<.P,.S>;
            (@.ITEM+.J)<.P,.S>;
            (@@.ITEM+.J)<.P,.S>;
            (@(@.ITEM+1)+.J)<.P,.S>
          TES;

STRUCTURE VECTOR2[I]=
        [2*I](.VECTOR2+2*.I)<0,36>;

MACRO   BASE=0,0,0,18$,
        RLINK=1,0,0,18$,
        LLINK=1,1,0,18$,
        ITEMSIZE=1,0,18,18$,
        NXTRLINK=2,0,0,18$,
        NXTLLINK=2,1,0,18$,
        PRVRLINK=3,0,0,18$,
        PRVLLINK=3,1,0,18$;

GLOBAL VECTOR2 SPACE[LOG2MEMSIZE+1];

BIND VECTOR SIZE =
        PLIT(1^0,1^1,1^2,1^3,1^4,1^5,1^6,1^7,1^8,1^9,1^10,
             1^11,1^12);

MACRO   PARTNER(B1,B2,S)= ((((B1)-MEM<0,0>) XOR ((B2)-MEM<0,0>))
                           EQL .SIZE[S])$,
        REPEAT= WHILE 1 DO$,
        BASEADDR(B,S)= MEM[((B)-MEM<0,0>) AND NOT .SIZE[S]]<0,0>$,
        ERRMSG(S)= ERROR(PLIT ASCIZ S)$;
```

```
! SPACE-MANAGEMENT-ROUTINES
!----------------------------

FORWARD EMPTY,ERROR,LINK,DELINK,COLLAPSE;

GLOBAL ROUTINE GET(N)=

  !RETURNS THE ADDRESS OF AN ITEM OF SIZE 2**N

  BEGIN REGISTER ITEM R;
    IF .N LEQ 0 OR .N GTR LOG2MEMSIZE
      THEN ERRMSG('INVALID SPACE REQ');
    IF NOT EMPTY(SPACE[.N]<0,0>)
      THEN R[BASE]←DELINK(.SPACE[.N])
      ELSE
        BEGIN
          R[BASE]←GET(.N+1);
          COLLAPSE(.R[BASE]+.SIZE[.N],.N)
        END;
    R[ITEMSIZE]←.N;
    .R[BASE]
  END;


ROUTINE COLLAPSE(A,N)=

  !CALLED BY RELEASE AND GET TO ATTEMPT TO COMPACTIFY SPACE
  !IF ADJACENT ITEMS ARE FREE

  BEGIN MAP ITEM A; REGISTER ITEM L;
    REPEAT
      BEGIN
        L[BASE]←SPACE[.N]<0,0>;
        WHILE .L[RLINK] NEQ SPACE[.N]<0,0> DO
          IF PARTNER(.L[RLINK],.A[BASE],.N)
            THEN
              BEGIN
                A[BASE]←BASEADDR(DELINK(.L[RLINK]),.N);
                N←.N+1;
                EXITCOMPOUND[2]
              END
            ELSE L[BASE]←.L[RLINK];
        RETURN (A[ITEMSIZE]←.N; LINK(.A[BASE],.L[BASE]))
      END;
  END;

GLOBAL ROUTINE RELEASE(A)=

  !CALLED TO RELEASE ITEM A

  BEGIN
    MAP ITEM A;
    COLLAPSE(.A[BASE],.A[ITEMSIZE])
  END;
```

```
! SIMPLE-LIST-ROUTINES
!----------------------

ROUTINE DELINK(A)=

  !REMOVES ITEM A FROM THE LIST TO WHICH IT IS APPENDED

  BEGIN MAP ITEM A;
    A[PRVRLINK]←.A[RLINK]; A[NXTLLINK]←.A[LLINK];
    A[RLINK]←A[LLINK]←.A[BASE]
  END;

ROUTINE LINK(A,TOO)=

  ! INSERTS ITEM A INTO A LIST IMMEDIATELY AFTER THE ITEM TOO

  BEGIN
    MAP ITEM A:TOO;
    A[LLINK]←.TOO[BASE]; A[RLINK]←.TOO[RLINK];
    TOO[NXTLLINK]←TOO[RLINK]←.A[BASE]
  END;


ROUTINE RELINK(A,TOO)=

  ! REMOVES ITEM FROM ITS PRESENT LIST AND INSERTS IT AFTER TOO

  LINK(DELINK(.A),.TOO);


ROUTINE EMPTY(L)=

  !PREDICATE INDICATING EMPTY LIST

  BEGIN MAP ITEM L;
    .L[BASE] EQL .L[RLINK]
  END;
```

```
! QUEUE-HANDLING-ROUTINES
!--------------------------

MACRO    QHDR=ITEM$;

MACRO    ENTER=1,2,0,18$,
         REMOVE=1,2,18,18$;


GLOBAL ROUTINE ENQ(A,Q)=

  ! ENTERS ITEM A ON QUEUE Q ACCORDING TO THE INSERTION DISCIPLINE
  ! EVOKED BY Q'S ENTER ROUTINE

  BEGIN
    MAP QHDR Q;
    RELINK(.A,(.Q[ENTER])(.Q[BASE],.A))
  END;


GLOBAL ROUTINE DEQ(Q)=

  ! REMOVES AN ITEM FROM QUEUE Q ACCORDING TO THE REMOVAL DISCIPLINE
  ! EVOKED BY Q'S REMOVE ROUTINE

  BEGIN
    MAP QHDR Q;
    DELINK((.Q[REMOVE])(.Q[BASE]))
  END;


! MISC SERVICE ROUTINES
!-----------------------


ROUTINE ERROR(A)=
  BEGIN MACHOP TTCALL=#051;
    TTCALL(3,.A)
  END;


ROUTINE INITIALIZE=

  !INITIALIZES THE SPACE MANAGEMENT DATA

  BEGIN REGISTER ITEM X;
    X[BASE]←MEM<0,0>;
    X[RLINK]←X[LLINK]←SPACE[LOG2MEMSIZE]<0,0>;
    X[ITEMSIZE]←LOG2MEMSIZE;
    DECR I FROM LOG2MEMSIZE-1 TO 0 DO
      SPACE[.I]←(SPACE[.I]+1)<0,36>←SPACE[.I]<0,0>;
    SPACE[LOG2MEMSIZE]←(SPACE[LOG2MEMSIZE]+1)<0,36>←MEM<0,0>
  END;
```

```
! EXAMPLES OF VARIOUS QUEUE MODELS
!---------------------------------

! LIFO QUEUE
!------------

ROUTINE LIFOREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;


ROUTINE LIFOENTER(Q,A)=
  BEGIN
    MAP QHDR Q;
    .Q[BASE]
  END;



! FIFO QUEUE
!------------


ROUTINE FIFOREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;



ROUTINE FIFOENTER(Q,A)=
  BEGIN
    MAP QHDR Q;
    .Q[LLINK]
  END;
```

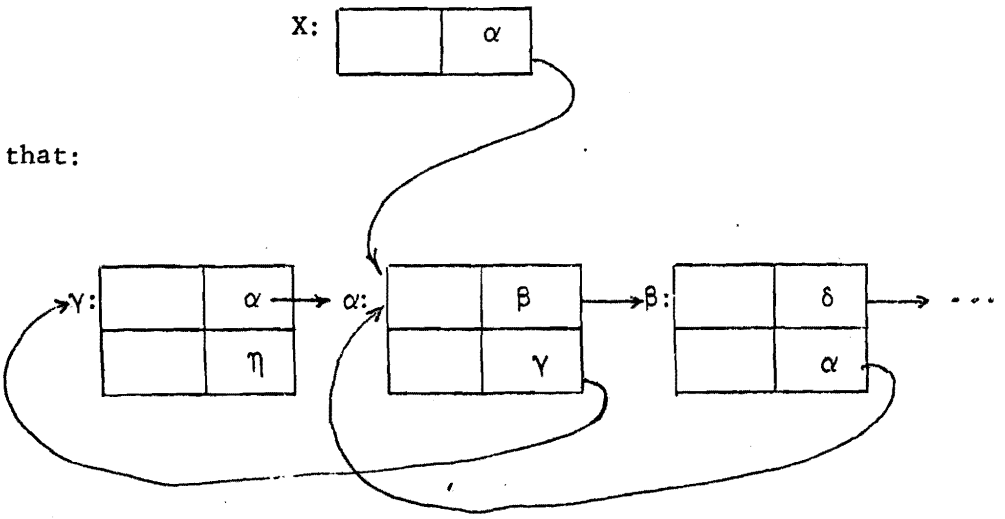
```
! PRIORITY QUEUE
!----------------


MACRO    PRIORITY=1,1,18,18$;

ROUTINE PRIREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;


ROUTINE PRIENTER(Q,A)=
  BEGIN
    MAP QHDR Q; MAP ITEM A; REGISTER ITEM L;
    IF EMPTY(.Q[BASE]) THEN RETURN .Q[BASE];
    L[BASE]←.Q[LLINK];
    UNTIL .L[PRIORITY] GEQ .A[PRIORITY] DO
      L[BASE]←.L[LLINK];
    .L[BASE]
  END;


END ELUDOM
```

Comments on the Use of Bliss in the Implementation

(1) The structure ITEM is particularly interesting and perhaps at first a bit obscure.

To illustrate, consider a variable X structured by item:

Assuming that the right half of X contains α:



and that:

Then:

.X[BASE] ≡ α	.X[NXTRLINK] = δ
.X[RLINK] ≡ β	.X[NXTLLINK] = α
.X[LLINK] ≡ γ	.X[PRVRLINK] = α
	.X[PRVLLINK] = η

The structure ITEM uses the "constant case" expression to distinguish between the pointer, the pointee, and the pointee's predecessor and successor.

(2) The structure VECTOR2 has a size expression [2*I] which is used in the allocating declaration:

```
GLOBAL VECTOR2 SPACE[LOG2MEMSIZE+1];
```

(3) Since the addresses of the 'remove' and 'enter' routines are stored in the queue header, the expression

```
(.Q[REMOVE])(.Q[BASE])
```

is a call of the routine whose address is .Q[REMOVE] and passes it to the base address of the queue or its parameter.

(4) The macro 'REPEAT = WHILE 1 DO' defines an infinite loop - its only exit is defined by the RETURN expression in its body.

(5) Notice the 'BIND VECTOR SIZE = PLIT(1↑0,1↑1,1↑2,...' in the space allocator. The value of SIZE is a pointer to this sequence of values, and in particular the value of '.SIZE[.N]' is 2^N.

EXAMPLE 3: DISCRIMINATION NET

Contributor: D. Wile

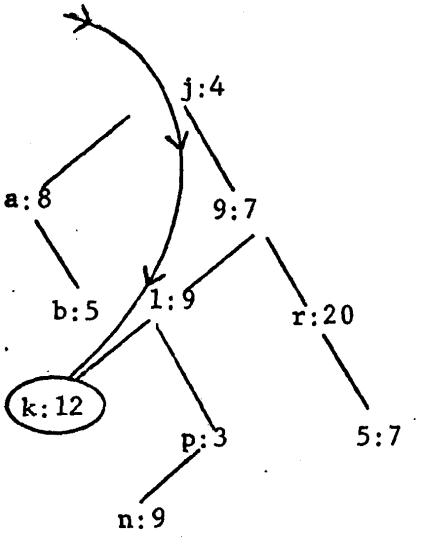
A discrimination net is a mechanism used to associate "information" with "names". The net is actually a tree, each node of which consists of a name and the information associated with that name, as well as a set of pointers to other nodes. To look up a name in the net we start at the root node and see if the name in the node matches our target name. If it does, we return the associated information.

Otherwise, we use a "discrimination function" which determines which subnode to examine next (usually as a function of the target name and the name of the current node). If there is no corresponding subnode, a new node must be created.

For example, a binary net (two subnodes/node) with a discrimination function which chooses the left branch if the target name is alphabetically smaller than the name in the node, is illustrated below:

Name: j, 9, 1, a, b, r, p, n, s, k

Inf: 4, 7, 9, 8, 5, 20,3, 9, 7, 12



In the implementation which follows, there are three globally defined routines:

1. DSCINIT (String address) -- returns a pointer to the information field of the node associated with the string. This must be called first to initialize the net. (The information field will be zeroed when the node is new.)

2. DSCLKP (String address) -- the "lookup" routine. Value returned as above.

3. DSCPNAME (Information field address) -- returns a pointer to the print name associated with the particular information field.

The implementation is designed to allow the user to create a module somewhat "tailored" to his needs. The module is created by passing:

1. the estimated number of entries to be inserted into the table;
2. the average number of words each name will occupy;
3. the number of words in the "information field";
4. the number of subnodes of each node (e.g., binary example above, 2);
5. a string which executes an error routine

in that order, to a macro "DSCRIMINET". Two macros must be defined previous to the DSCRIMINET expansion:

1. DSCIMINATE (Target string address, current node string address) must have a value of -1 if the strings match. Otherwise, its value must be between 0 and 1 less than the number of subnodes.

2. DSCCOPY (To address, From address) copies the string from the "from address" to the "to address", returning the number of words occupied by the copy.

```
MODULE NET(STACK=GLOBAL(STABK,#400))=
BEGIN
MACRO
  DSCRIMINET(MAXNUMENT,AVNAMESIZE,INFSIZE,NOSUBNODES,ERROR)=

    BEGIN
      %N.B.: ALL VECTOR ACCESSES ARE INDIRECT THROUGH THE BASE%
      STRUCTURE VECTOR[I]=(@.VECTOR+.I)<0,36>;

      % NET SPACE ALLOCATION, STRUCTURE DEFINITION AND
        INITIALIZATION DEFINITIONS %
      BIND TABLELEN=MAXNUMENT*((NOSUBNODES+1)/2+INFSIZE+AVNAMESIZE);
      OWN BASENODE[TABLELEN];
      BIND MAXADD=BASENODE+TABLELEN;

      BIND SUBNODE=0, INF=1, PNAME=2,
           INFOFFSET=(NOSUBNODES+1)/2,
           PNAMEOFFSET=INFOFFSET+INFSIZE;

      STRUCTURE NODE[SUBFIELD,INDEX]=CASE .SUBFIELD OF
        SET .NODE[.INDEX*(-1)]<IF .INDEX THEN 18,18>;
            .NODE[INFOFFSET];
            .NODE[PNAMEOFFSET] TES;

      GLOBAL ROUTINE DSCPNAME(INFPOS)=
        (.INFPOS+INFSIZE)<0,36>;

      OWN NODE NEXTCELL;

      ROUTINE INITNODE(CELL,STRING)=
        BEGIN
          DECR I FROM PNAMEOFFSET-1 TO 0 DO CELL[.I]=0;
          IF MAXADD LEQ (NEXTCELL=.NEXTCELL+PNAMEOFFSET+
            (MAP NODE CELL; DSCCOPY(CELL[PNAME],.STRING)))

            THEN ERROR ELSE .CELL
        END;

      GLOBAL ROUTINE DSCINIT(STRING)=
        BEGIN
          LOCAL NODE RETVAL;
          NEXTCELL=BASENODE;
          RETVAL=INITNODE(BASENODE,.STRING);
          RETVAL[INF]
        END;

      ROUTINE NEWCELL(STRING)=INITNODE(.NEXTCELL,.STRING);

      % THE LOOKUP ROUTINE ITSELF %
      GLOBAL ROUTINE DSCLKP(STRING)=
        BEGIN
          LOCAL DISCIND, NODE CURRENT:NEXT;
          NEXT=BASENODE;
```

```
            DO
              BEGIN
                CURRENT←.NEXT;
                IF (DISCIND←DSCIMINATE(.STRING,CURRENT[PNAME])) LSS 0
                  THEN RETURN CURRENT[INF];
                NEXT←.CURRENT[SUBNODE,.DISCIND]
              END

             UNTIL .NEXT EQL 0;

            NEXT←CURRENT[SUBNODE,.DISCIND]←NEWCELL(.STRING);
            NEXT[INF]
          END;
      END;$;


    ROUTINE DSCIMINATE(L,R)=
      BEGIN
        STRUCTURE VECTOR[I]=(@.VECTOR+.I)<0,36>;
        INCR I FROM 0
          DO BEGIN
               BIND LEFT=.L[.I], RIGHT=.R[.I];
               IF LEFT NEQ RIGHT THEN EXITLOOP (LEFT LSS RIGHT);
               IF (LEFT AND #376) EQL 0 THEN EXITLOOP -1
             END
      END;

    ROUTINE DSCCOPY(INTO,FRO)=
      BEGIN
        STRUCTURE VECTOR[I]=(@.VECTOR+.I)<0,36>;
        INCR I FROM 0 DO
          IF ((INTO[.I]←.FRO[.I]) AND #376) EQL 0
            THEN EXITLOOP .I+1
      END;

    EXTERNAL ERROR;
    DSCRIMINET(500,3,1,2,ERROR(PLIT 'LOOKUP TABLE OVERFLOW'))

  BEGIN
    BIND NAMES=PLIT(
      PLIT ASCIZ 'FIRSTNAME',
      PLIT ASCIZ 'SECOND',
      PLIT ASCIZ 'SS',
      PLIT ASCIZ 'A LONGISH NAME',
      PLIT ASCIZ 'L',
      PLIT ASCIZ '77788()34');

    EXTERNAL DSCLKP, DSCINIT;
    DSCINIT(PLIT 'ZEROTH NAME')←-3;
    INCR I FROM 0 TO .NAMES[-1]-1 DO DSCLKP(.NAMES[.I])←.I;
    INCR I FROM 0 TO .NAMES[-1]-1 BY 2 DO DSCLKP(.NAMES[.I])←.I+1↑35;
  END;
  END ELUDOM;;
```

Notes on the Implementation

The Bliss module above implements the example described at the beginning of this section. The test program portion of the module simply initializes the table, inserts the six strings in the plit into the table (associating as information, the index in the plit), and runs through the evenly indexed items in the plit, turning on the sign bit in the information word.

Of interest:

1. The vector structure (which defaults as the structure for all unmapped variables and expressions) is redefined "indirectly"; this is fairly dangerous in any program, and represents an after-the-fact programming decision.

2. The physical structure of the table is kept independent of the logical structure as used by the lookup routine; no reference is made from the lookup routine to the structure other than through the structured nodes.

3. The binds, structures, own declarations and even the initialization function - requiring knowledge of the physical structure are kept grouped and separate. Note, for example, that INITNODE uses both a vector mapping on contiguous fields of CELL and the NODE structure.

4. The physical structure of the tree is kept isolated from the user of the routines to the extent that only knowledge

that the mechanism is associative is of importance -- the particular lookup algorithm and storage management are independent of the functional use of the module.

5. Bliss programming "tricks":

   a. Use of the constant case expression for sub-fields of structures (NODE in this case);
   b. Default use of 0 for the omitted else in the structure case defining the SUBNODE field;
   c. CELL remapped in the INITNODE routine to take advantage of knowledge of the physical layout of the NODE's storage.
   d. "Dynamic" binds of LEFT and RIGHT inside the loop in the test version discrimination function;
   e. The bind to a plit (of NAMES) in the test portion, to prevent duplicate storage allocation for the twice-used plit;
   f. Stores into routine cells in the test program loops;
   g. Use of the plit length word preceding the plit (NAMES[-1]).

Programming Without the GOTO

William A. Wulf

PROGRAMMING WITHOUT THE GOTO

William A. Wulf*
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa., U.S.A.

*This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) 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.

ABSTRACT

It has been proposed, by Dijkstra and others, that the use of the goto statement is a major contributing factor in programs which are difficult to understand and debug. This suggestion has met with considerable skepticism in some circles since goto is a control primitive from which a programmer may synthesize other, more complex, control structures which may not be available in a given language. This paper analyzes the nature of control structures which cannot be easily synthesized from simple conditional and loop constructs. This analysis is then used as the basis for the control structures of a particular language, Bliss, which does not have a goto statement. The results of two years of experience programming in Bliss, and hence without goto's, are summarized.

INTRODUCTION

In 1968 E. W. Dijkstra suggested, in a letter to the editor of the Communications of the ACM [1], that use of the goto construct of Algol was undesirable, and in fact was bad programming practice. The rationale behind this suggestion was simply that it is possible to use the goto in ways which obscure the logical structure of a program, thus making it difficult to understand, debug, and, ultimately, to prove its correctness. Of course, not all uses of the goto are obscure, but the conjecture is that these situations are adequately handled by existing conditional (e.g., the if-then-else) and looping (for-do) constructs.

This paper presents an analysis which lead to the design of the control features of Bliss [5], an implementation language designed at Carnegie-Mellon University. This analysis reveals that the Algol conditional and looping constructs are, while adequate, not convenient when the goto is eliminated. The control features of Bliss are described and some comments are made concerning our experiences using a goto-less, Algol-like language.

Before proceeding it is worth noting an additional benefit of removing the goto - a benefit which the author did not fully appreciate until the Bliss compiler was designed - that of code optimization. It is clear that the presence of goto in a block-structured language with dynamic storage allocation forces a certain amount of run-time support (and overhead) associated with the possibility of jumping out of blocks and procedure bodies. Eliminating the goto obviously removes this overhead. Far more important, however, is the fact that the scope of a control environment

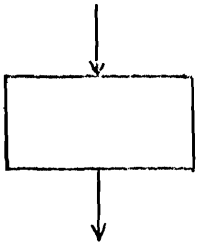
is statically defined in a program without goto's. The Fortran-H compiler [2], for example, does considerable analysis and achieves a less perfect picture of the overall control structure of a program than that implicit in the text of a Bliss program. Since analysis of control flow is prerequisite to any form of global optimization, this benefit of eliminating the goto must not be underestimated.

It is not surprising that a language can be devised which does not use the goto construct since: (1) several of the formal systems of computability theory, e.g., recursive functions and the λ-calculus, do not contain the concept; (2) LISP does not use it; and (3) Van Wijgaarden [3], in attempting to define the semantics of Algol, eliminated labels and goto's by systematic substitution of procedure bodies and calls. Thus, the question is not whether it is possible to remove the goto, only whether it is desirable. In particular there is considerable suspicion among programmers that the advantages described by Dijkstra are outweighed by inconvenience, and possibly by inefficiency (duplicate code, etc.).

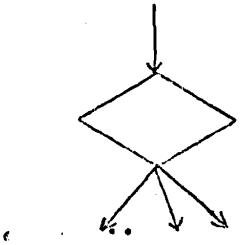
The goto may be viewed as a control primitive with which a programmer synthesizes more complex control structures. In this context Dijkstra's arguments can be phrased in terms of this primitive having "unwanted generality". The principle concern of this paper is to investigate alternative primitives which are equally convenient for the things which programmers actually do.

ANALYSIS

In order to determine the nature of the control primitives to substitute for the goto, we shall first consider the nature of programs which use the goto and which cannot be easily built from simple conditional and looping constructs. To do this we will use a flow chart representation of programs. Flow charts are convenient for this because of the explicit way in which control is manifest in them. We assume two basic blocks from which our flow charts are to be built - process blocks and n-way conditionals.

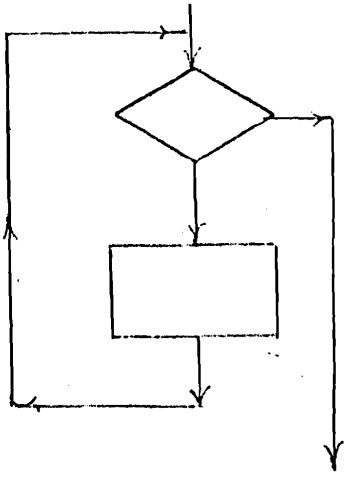


process box

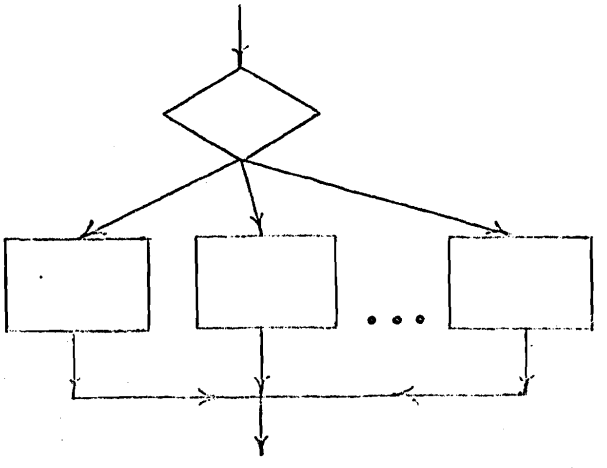


n-way conditional

These boxes are connected by directed line segments in the usual way. We shall further be interested in two special "goto-less" constructions built from these components - simple loop and n-way "case" constructs.



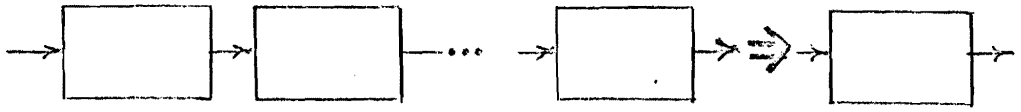
simple loop



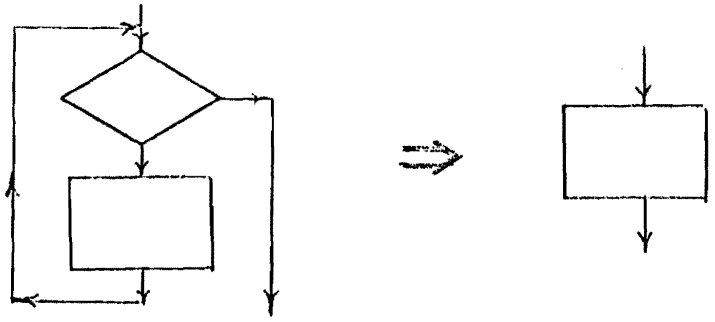
case

We consider these two forms "goto-less" since they contain a single entry point and a single exit point and hence could have reasonable corresponding syntactic constructs in some higher-level language (and indeed do).* Now, consider three transformations:

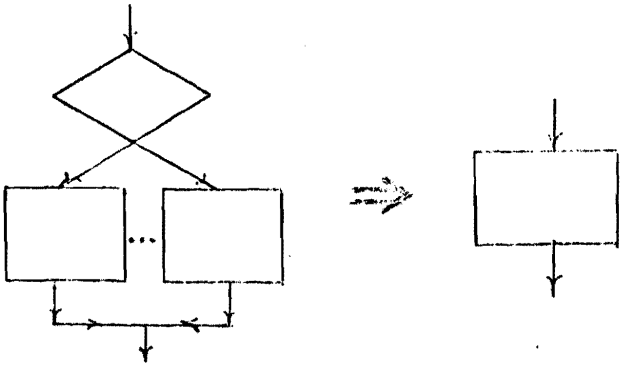
1. any linear sequence of process boxes may be transformed into (or replaced by) a single process box



2. any simple loop may be replaced by a process box



3. any n-way case construct may be replaced by a process box



*The simple loop considered here clearly does not correspond to all possible varients of initialization, test before or after the loop body, etc. These varients would not change the arguments to follow in any essential way and hence have been omitted.

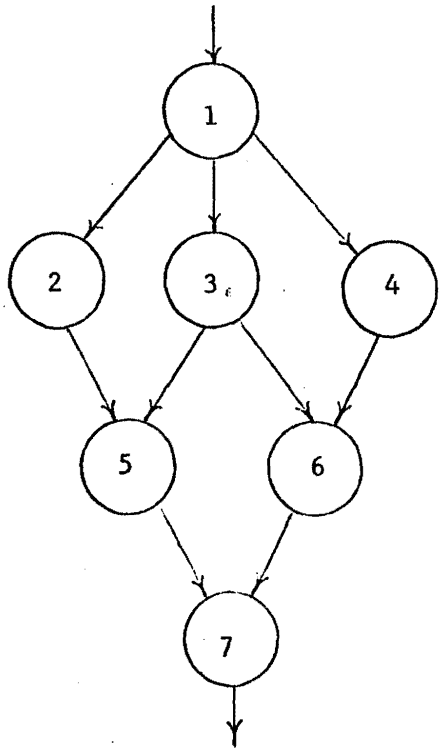
Any graph which may be derived from a given graph by a sequence of these transformations we shall call a "reduced" form of the original graph. A graph which has a reduced form consisting of a single process box we shall call a simple "goto-less" graph. The sequence of transformations is said to define a set of nested "control environments".

Not all graphs are of this type; these are of special interest to us since they typify the class of control structures which cannot be realized by simple conditional and looping constructs. In looking at such graphs we are principally interested in their "minimal irreducible form"; that is, a reduced form to which no more transformations of the type described can be applied. Examination of these graphs will both reveal techniques for deriving simple goto-less graphs from them, and also provide insight leading to the control primitives to be described later.

Before proceeding it is perhaps instructive to remark briefly on Dijkstra's objections to the goto in terms of this characterization of programs. By definition, a goto-less program (flow chart) is susceptible to a sequence of simple transformations which reduces it to a single process box. This sequence can serve as guide to understanding and/or proving the correctness of the program. Imagine a sequence of graphs, derived from the original, in which each is like its predecessor except:
(1) the correctness of the replaced construct has been verified, and
(2) the new process box contains a more macroscopic description of what the replaced portion does (rather than the details of how it is done).
This sequence forms both a proof of the validity of the entire original program as well as documentation of what it does (at many levels of detail).

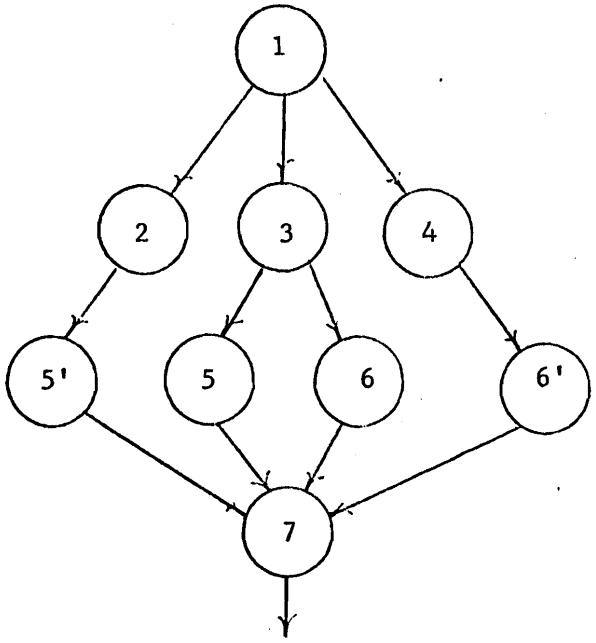
This is not to say that programs that use goto cannot be understood or proved correct [6], only that programs with this structure permit a specific methodical approach to understanding and proof.

Now, returning to an analysis of programs which use goto, consider two cases - those with loops and those without. Programs without loops have, at most, a lattice-like structure. For example, consider the following irreducible form (in this example, and the remainder of the paper, we shall use circles to represent sub-graphs whose fine-structure we choose to ignore):

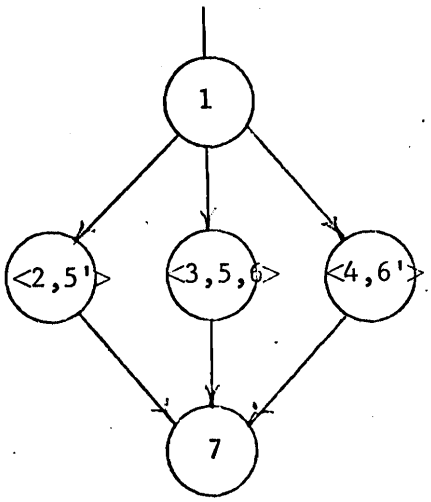


Brief consideration of such graphs reveals that it is always possible to construct a new graph using only the goto-less primitives which are similar to the original graph except for a finite number of "node splittings"

(i.e., creation of duplicates of existing nodes in separate control paths). This follows from the observation that, since there are no loops, there are at most a finite number of paths through the graph and each node occurs on only finitely many of them. Hence at most a finite number of replications of each node will guarantee that each node will occur on only one path. For example, the graph above becomes:



And this graph can now be transformed by collapsing <2,5'>, <3,5,6>, and <4,6'> into:



This is one of the primitive forms and may itself be collapsed - and hence is a goto-less program.

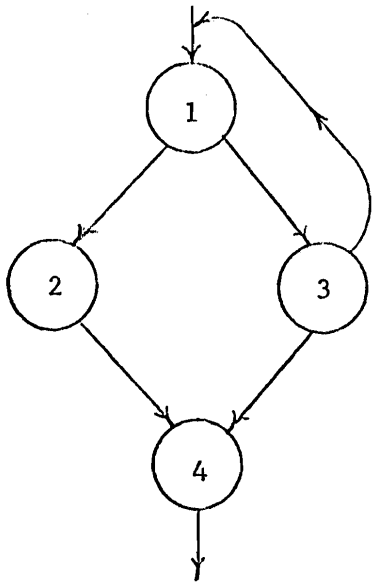
Node splitting is something which we would like to avoid since it involves duplicating code. Nevertheless, node splitting is one technique by which an existing program utilizing the goto may be converted into one which does not. A second technique, which also might have been used above, will be discussed in conjunction with loops below.

The second major case to be considered is that of irreducible graphs involving a loop. Of these we can note that such loops must involve more than one entry or exit point. Otherwise the loop would be reducible.*

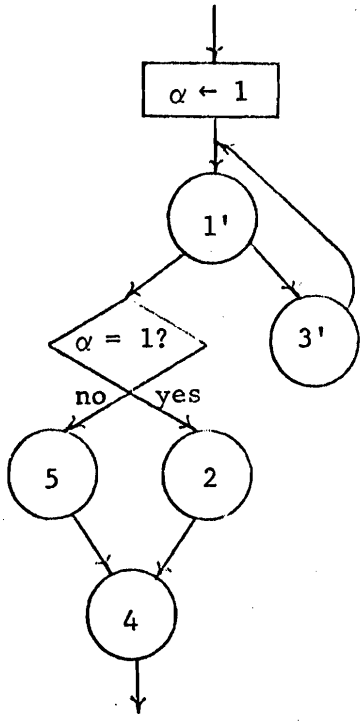
Floyd and Knuth [4] have proven (using flow charts as specifications for regular expressions) that node splitting is not an adequate technique for deriving goto-less graphs from irreducible ones in the presence of multiple entry/exit loops.

That node splitting is inadequate becomes clear by simply observing that the number of paths leading from the "second" exit point is unbounded. Therefore no finite number of replications of this node is sufficient, and we must search for another technique. Consider the following irreducible program:

*We reiterate our earlier footnote - we have only considered one form of simple loop - introducing varients on the initialization or relation of the test to the loop body would not affect these arguments in any essential way.



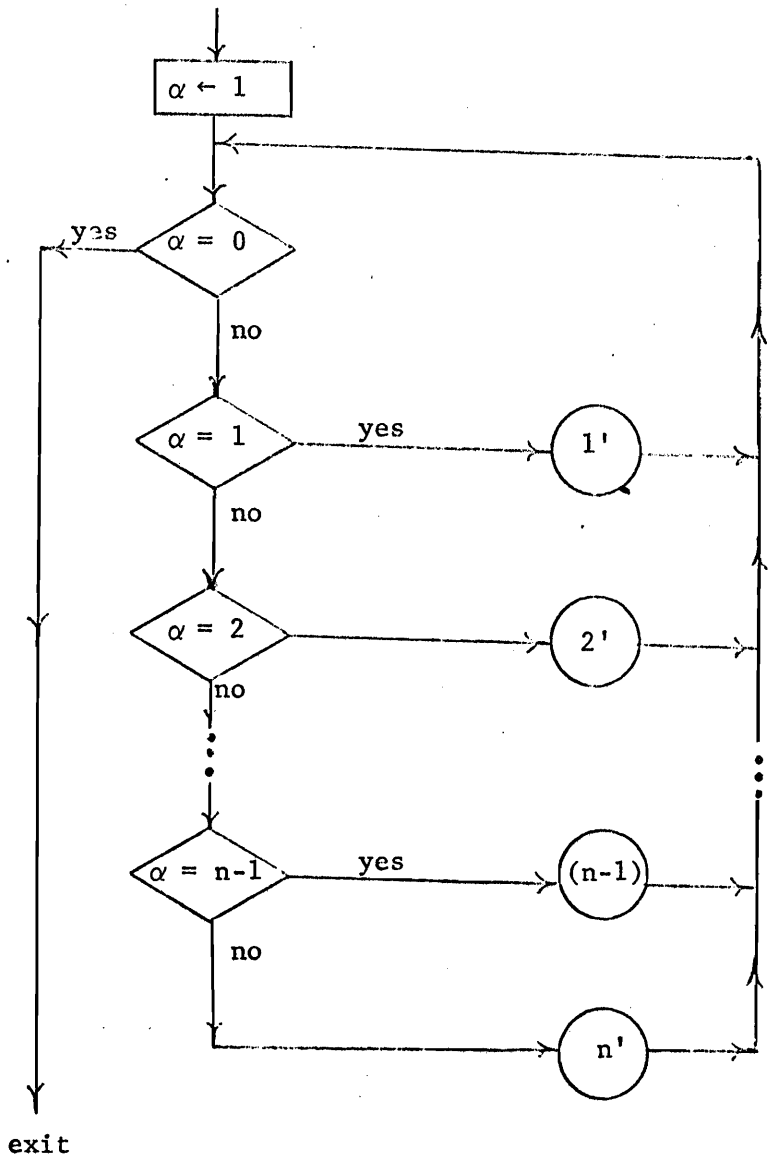
Notice that there are two exit paths from the ①-③ loop - that leading from ① to ② to ④ and that leading from ③ to ④ directly. This is a simple example of a program where node splitting will not work. However, one can introduce a new variable, call it α, and obtain the following graph:



In this graph the node (1') is like node (1) except that the exit condition of the loop has been augmented with "or $\alpha = 0$" and node (3') is like node (3) except that the exit to node (4) has been replaced by the operation "$\alpha \leftarrow 0$". Node (5) is the null operation. Conceptually what we have done is to introduce a variable which behaves as a "program counter" and which, when the loop terminates, specifies whether or not it is necessary to execute (2) .

That the technique illustrated above is completely general may be seen easily. Consider any graph with nodes labeled (i) , (2) , ..., (n) . Now construct a new graph as follows:

1. if (i) is a process box construct (i') by adding to (i) "$\alpha \leftarrow k$" where (k) is the successor of (i) .
2. if (i) is a decision box, then replace it by a process box of the form "$\alpha \leftarrow \epsilon$", where ϵ is an expression which dynamically evaluates to the appropriate successor label.
3. consider all exit points as labeled by (0)
4. construct the following graph



As with node splitting, this technique is odious because of the implied inefficiency. But also, it is a technique which may be applied to convert any existing programs with gotos into ones without them. And, in particular, the techniques may be applied locally to irreducible sub-graphs.

The Bliss Control Structure

The previous section points out the nature of programs which may be constructed with only conditional and looping constructs - and those which cannot be constructed without duplicating some nodes or adding dummy variables, etc. The present section addresses itself to the question of whether the class of constructs in a practical language (which will not contain an explicit goto) should be extended beyond simple conditional and looping facilities. And, if the decision is to extend the class, then what should the extensions be? The answer to the first of these questions depends in part on a judgment as to the frequency with which multiple exits from loops, etc., are used, and in part on the answer to the second question. Whether to add constructs or not depends upon whether it can be done in such a way as to preserve the structural advantages which prompted us to consider a goto-less language in the first place. Hence we must answer the question of a specific language proposal. Part of this section will be devoted to a description of the facilities in Bliss to give some background for discussing this question.

Note that we are principally interested in programs which are initially written in such a goto-less notation rather than in translating existing programs into the notation. Consequently, we are willing to accept some restrictions on what can be written - so long as the "common" things are expressed conveniently. Even the goto is not completely general in most languages - one may not jump into the scope of a DO statement nor out of a subroutine in FORTRAN, and jumping into the middle of a block from outside it is prohibited in Algol. Neither of these restrictions is a serious one in practice.

The three "problem areas" discussed in the first section were: (1) lattice-like decision structures, (2) multiple entry points to a loop, and (3) multiple exits from a loop. Without any hard evidence at our disposal we are left with only our intuition and experience to weight the importance of these constructs. In particular, the author believes that (1) and (3) are both quite important, and only one subcase of (2) is important - namely, that case involving selection of one of several initialization sequences. One might make a different evaluation and arrive at a different set of facilities than those to be described below.

The first aspect of the Bliss control structure is simply the fact that it is a block-structured "expression language". That is, every executable construct, including those which manifest control, is an expression and computes a value. There are no statements in the sense of Algol or PL/I. Expressions may be concatenated with semicolons to form expression sequences. The value of an expression sequence is that of its last (rightmost) component expression and is evaluated in strictly left-to-right order. Thus ";" may be thought of as a dyadic, left associative operator whose value is simply that of its righthand operand. A pair of symbols begin and end, or left and right parentheses, may be used to embrace such an expression sequence and convert it into a simple expression. A block is merely a special case of this construction which happens to contain declarations, thus the value of a block is defined to be the value of its constituent expression sequence.

The fact that Bliss is an expression language is relevant to the goto issue in the following way: the most general method described in the first

section for translating programs into goto-less form was that involving a dummy variable which explicitly indicates the successor. The value of an expression (a block, for example) forms a natural implicit node of expressing this idea. This will be illustrated after some of the explicit control expressions have been discussed.

There are six explicit control expressions in Bliss: conditional, loop, case-select, function, co-routine, and escape. We have avoided consideration of subroutines in the previous material and so shall omit functions and co-routines from this discussion.

The conditional expression

$$\underline{\text{if}}\ \epsilon_1\ \underline{\text{then}}\ \epsilon_2\ \underline{\text{else}}\ \epsilon_3$$

is defined to have the value of the expression ϵ_2 just in the case that ϵ_1 evaluates to the Bliss representation of true and has the value of ϵ_3 otherwise. The abbreviated form "$\underline{\text{if}}\ \epsilon_1\ \underline{\text{then}}\ \epsilon_2$" is considered to be identical to "$\underline{\text{if}}\ \epsilon_1\ \underline{\text{then}}\ \epsilon_2\ \underline{\text{else}}\ 0$".

The conditional expression provides two-way branching, the case and select expression provide more general n-way branching:

$$\underline{\text{case}}\quad e_0,e_1,\ldots,e_k\ \underline{\text{of set}}\ \epsilon_0;\ \epsilon_1;\ \ldots;\ \epsilon_n\ \underline{\text{tes}}$$

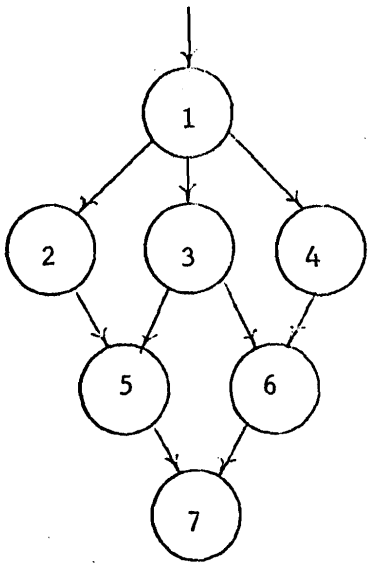
$$\underline{\text{select}}\quad e_0,e_1,\ldots,e_k\ \underline{\text{of set}}\ \epsilon_0{:}\epsilon_1;\ \epsilon_2{:}\epsilon_3;\ \ldots\ ;\ \epsilon_{2n}{:}\epsilon_{2n+1}\ \underline{\text{tesn}}$$

The case expression is executed as follows: (1) all of the expressions $e_0,\ldots,e_k$ are evaluated, (2) the value of each e_i $(0 \le i \le k)$ is, in turn from left to right, used as an index to choose one of the ϵ_j's $(0 \le j \le n)$ to be executed. Obviously, each of the e_i's is constrained

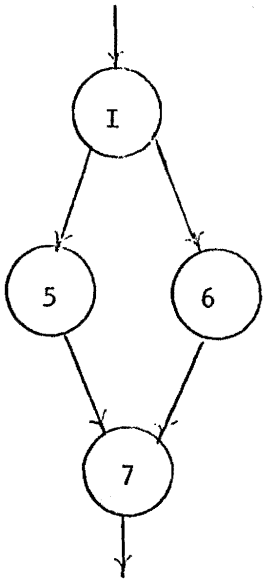
to lie in the range $0 \leq e_i \leq n$ if one of the ϵ's is to be executed. In the current implementation if $e_i = -1$ none of the ϵ's will be executed and execution is undefined for all other values of e_i. The value of the entire case expression is ϵ_{e_k}. The special case where k=1 is of special interest and has appeared in several other languages, ALGOL-W and EULER, for example.

The select expression is similar to the case expression except that the e_i's are not used as indices. Rather, the e's are used in conjunction with the ϵ_{2j}'s to choose among the ϵ_{2j+1}'s. Execution proceeds as follows: (1) all of the e_i's are evaluated, (2) ϵ_0 is evaluated, (3) if the value of ϵ_0 is identical to the value of one (or more) of the e's then ϵ_1 is executed, (4) ϵ_2 is evaluated, (5) if the value of ϵ_2 is identical to the value of one (or more) of the e's then ϵ_3 is executed, etc. The value of the entire select expression is simply that of the last ϵ_{2j+1} to be executed - or -1 if none of them is executed.

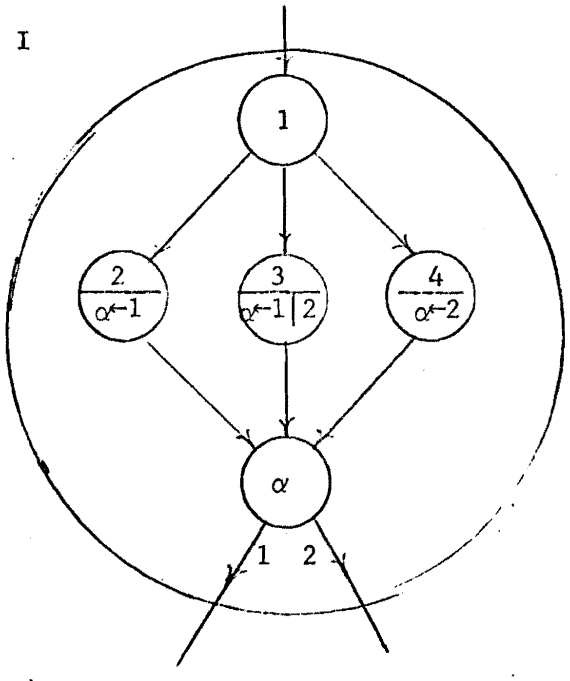
The utility of the fact that Bliss is an expression language may be illustrated using the case expression in an earlier example, namely the flow chart:



This graph may be thought of as actually of the form



where (I) is formed from (1), (2), and (3) as follows:



Which means that one might write in (pseudo) Bliss:

```
case case (1) of set  ((2);0);((3); 0V1);((4);1) tes of
        set(5); (6) tes;
(7)
```

This provides a neat, conceptually simple, and efficient alternative to node splitting.

Returning now to the discussion of Bliss control forms, the loop expressions imply repeated execution (possibly zero times) of an expression until a specific condition is satisfied. There are several forms, some of which are:

```
while ϵ1 do ϵ
do ϵ while ϵ1
incr <id> from ϵ1 to ϵ2 by ϵ3 do ϵ
```

In the first form the expression ϵ is repeated so long as ϵ_1 satisfies the Bliss definition of true. The second form is similar except that ϵ is evaluated before ϵ_1 thus guaranteeing at least one execution of ϵ. The last form is similar to the familiar "step...until" construct of Algol, except (1) the control variable, <id>, is local to ϵ, and (2) ϵ_1,ϵ_2 and ϵ_3 are computed only once (before the first evaluation of the loop body, ϵ). Except for the possibility of an escape expression within ϵ (see below) the value of a loop expression is uniformly taken to be -1. The particular choice of -1 as the value of a loop expression is not important except that: (1) it is uniform, and (2) there are some small advantages to this choice in connection with the definition of the case expression and zero origin data structures.

The control mechanisms described above are either similar to, or only slight generalizations of, the conditional and loop constructs of many other languages. Of themselves they do not solve the problems discussed in the first section. Another mechanism is needed - that mechanism is called the escape expression. An escape expression provides a highly structured form of forward branch. The branch is constrained to terminate coincidentally with the terminus of some control environment in which the escape expression is nested. The general form of an escape expression is

<escapetype> <levels> <expression>

where <escapetype> is one of the (reserved) words listed below and <levels> is either an integer enclosed in square brackets, e.g., "[3]", or else is empty (which implies [1]).

exitblock	exitcase
exitcompound	exitselect
exitloop	exit
exitconditional	return

An escape expression causes control to immediately exit from a specified control environment (a block, a compound, or a loop, for example) skipping any subsequent expressions in that environment. The <levels> construct permits exit from several nested loops, for example, with a single exitloop expression. The <expression> value in an escape expression defines the value of the environment from which control passes.

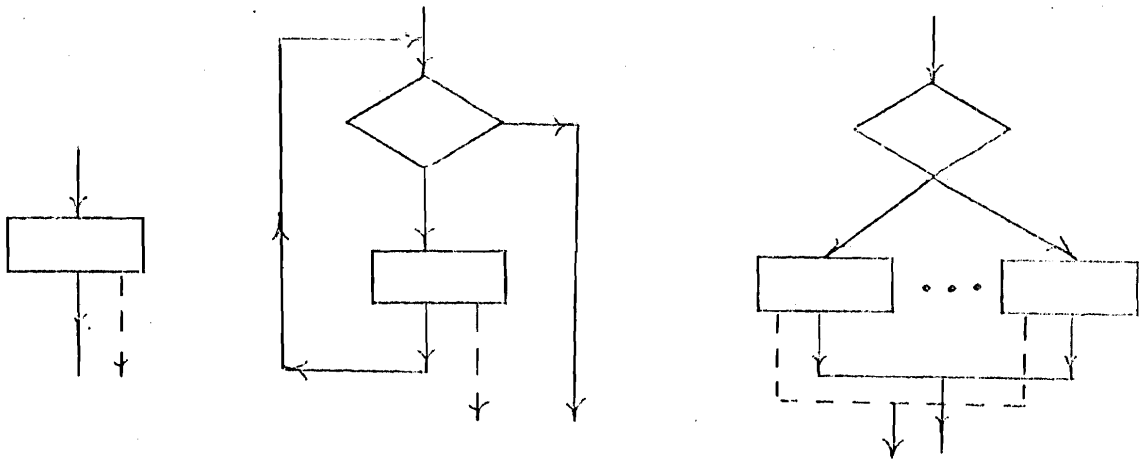
The use of the escape expression is illustrated by a typical problem involving multiple exits points from a loop. Suppose a vector, X, is to be searched for a value, x. If an element of X is equal to x, then the variable, k, is to be set to the index of this element. If no element of X is equal to x, then the value of x is to be inserted after the last element of X and k set to this index. Supposing there are N elements currently in X. The following Bliss program* will perform this task.

```
if (k ← incr i from 1 to N by 1 do if X[i] = x then exitloop i) < 0
        then X[k ← N ← N+1] ← x;
```

We can now return to the original questions raised in this section. We know that the mechanisms are "adequate", but are they sufficiently convenient and do they preserve the desirable properties of goto-less-ness. The answer to the first of these questions lies principally in the experience of those who have used the language. These experiences are summarized in the next section and essentially answered in the affirmative. Some confidence that this is the case may be gained by simply viewing the escape mechanism as a specific device for handling multiple exit point loops, and viewing the decision to make Bliss an expression language as a specific tool for implementing the dummy variable technique. In fact, of course, both ideas are more general than this.

The second question, whether the Bliss structures retain the desirable properties of simpler goto-less notations, requires a little more consideration. First, it is only the escape mechanism which violates the goto-less criteria. Returning to the flow chart notations, we now think of our flow chart primitives as:

*Actually the given program is not Bliss, but the differences are not essential to the discussion of control.



where the dotted lines represent a potentially infinite set of flow lines one of which may be followed if the escape mechanism is invoked. Dotted flow lines are constrained to connect directly to the terminus of a control environment in which the initial point of the line is totally nested.

The previous set of transformations is still applicable if the dotted, "escape", flow lines are ignored and we are guaranteed that the escape lines will be totally enclosed at some stage in the reduction process. In this sense the desirable properties of goto-less graphs are retained. The simple technique for understanding a flow chart and proving its correctness is no longer possible, however, because control is no longer constrained to exit through a single path. Nevertheless, a similar technique is easily constructed. It simply must operate in more global contexts.

One can clearly apply the former style of reasoning to subgraphs from which no dotted lines emanate. After this has been done on all possible subgraphs attention must shift to as small a subgraph as possible which wholly contains its escape lines, and understanding be gained and verification done on this subgraph as a whole, and reduced as a whole. This may

or may not lead to the simpler form of graph, but in either case the process can be iterated.

Some Experiences

Bliss has been in active use for nearly two years and we have therefore gained considerable experience in programming without the goto - both in writing new programs and in translating previously existing ones. This experience includes several compilers, parts of an operating system, i/o support routines, as well as numerous applications programs. As one might expect, writing new programs presents no difficulty. Just as one adapts to the lack of recursion in Fortran or the inability to jump into the middle of an Algol block, one also adapts to the Bliss control structure. But it is not that one merely survives in this mode; quite the contrary. One develops a mode of thinking whihc is roughly the inverse of the reduction transformation sequence discussed in the first section. That is, one thinks, and writes from the more macroscopic to the most detailed levels. We have not conducted controlled experiments, but I am convinced that programmer productivity has significantly improved due to this enforced style of programming.

In some sense our experiences in translating existing programs are even more interesting than those in writing new ones. These latter experiences fall in two sharply defined categories - the times when it was easy and the times when it was hard. Most of the time it was easy, because most of the time programmers apparently use goto's in non-essential ways; that is, ways which mirror one or more of the constructs already in Bliss. On

the other hand, when the translation was difficult the real problem was understanding what the original programmer had intended the control structure to be. Once that was done, in every case (to my knowledge) there was a natural mode of expression in Bliss. There were surprisingly few cases where node splitting, or any of the other devices mentioned, were necessary. If we assume that the programs we have translated are representative, and I do not know that they are, then we must conclude that programmers do not use the generality of the goto.

We have found two aspects to the Bliss structure which are inconvenient and should be changed. One is a trivial syntactic change and is easily accomplished; the other is more fundamental. The "<levels>" construct in escape expressions embodies an important semantic notion, but the syntax should be changed. As a program is modified the number of levels through which an escape should execute may be changed - by the introduction of an additional block level, for example. One would like to indicate the target of the escape symbolically. Which is to say labels should be reintroduced as names of entire control environments. The other construct I should like to have is, intuitively, one which allows exit through several levels of subroutine call - either to a specific place or until a specified condition is met.

Whether or not a language includes the goto construct is immaterial. There are certain types of control flow which occur in real programs and if constructs are not explicitly provided for these then the goto must be provided so that the programmer may synthesize them for himself. The danger in permitting the goto is that the programmer will synthesize them

in weird and obscure ways. The advantages in eliminating the goto are that these same control structures will appear in regular and well-defined ways and consequently both the human reader and the compiler will do a better job of interpreting them.

References

[1] Dijkstra, E. W., "Goto Statement Considered Harmful", Letter to the Editor, Communications of the ACM, 11, 3 (March 1968).

[2] Lowery and Medlock, "Object Code Optimization", Communications of the ACM, 12, 1 (January 1969).

[3] Van Wijngaarden, A., "Recursive Definition of Syntax and Semantics", in Formal Language Description Languages for Computer Programming, (T. B. Steel, ed.), North-Hollan Publishing Co., Amsterdam, 1966.

[4] Knuth, Floyd, "Notes on Avoiding 'GOTO' Statements", Technical Report No. CS 148, Computer Science Department, Stanford University, January 1970.

[5] Wulf, et. al, "Bliss Reference Manual", Computer Science Department Report, Carnegie-Mellon University.

[6] King, J., "A Program Verifier", Ph.D. Dissertation, Carnegie-Mellon University, 1969.

Why the DOT?

William A. Wulf

WHY THE DOT?

The interpretation of the occurrence of identifiers in Bliss is different from that in most programming languages - and this difference has given rise to questions and suggestions from almost everyone who is first introduced to the language. The purpose of this memo is to, or at least attempt to, explain the reason for the chosen interpretation. The chosen interpretation is quite fundamental to the intent and structure of the language and was decided upon only after extensive, heated debate and is not merely a whim of the designers; to change it would do substantial violence to the language and could only be accomplished through the introduction of a large number of ad hoc rules if the other intentions of the language were to be preserved.

First let me review the interpretation, although I'm assuming some acquaintence with the language. An identifier is introduced into a Bliss program by a declaration; for example

```
own x;
```

There are scope rules as in Algol '60, but let's ignore them and assume that x is not re-declared at an inner block level. Now, anywhere in the scope of this declaration, independent of the context in which it occurs, an occurrence of the identifier is interpreted to mean a reference* to the memory cell allocated by the declaration. Thus the value of the expression "x+1" is one larger than the address of x rather than the value contained in the memory cell x. Thus, one may think of the occurrence

---

*A reference, or pointer, in Bliss is a fairly complex object, but for this discussion it is adequate to think of it merely as the address of a memory cell. The remainder of the discussion presumes this simple interpretation.

of an indentifier, x, as the occurrence of a literal (the address of x) where the value of the literal is bound at load (or possibly execution) time.

Clearly one wants to obtain the value stored in a memory cell as well as its address. For this purpose the unary dot, ".", operator is introduced. The value of the dot operator applied to an expression, ϵ, is that of the memory cell whose address is ϵ. Thus, ".x" is the value contained in the memory cell x, ".(x+1)" is the value of the memory cell whose address is one greater than that of x, "..x" is the value of the memory cell whose address is stored in the memory cell whose address is x (i.e., indirect addressing), etc.

Closely associated with the interpretation of identifiers and the dot operator is that of the store operator "←", which is also different from that usually given in the description of conventional languages (though not different from its implementation). The store operator is a dyadic, infix operator whose operands may be arbitrary expressions, say ϵ_1 and ϵ_2.

$$\epsilon_1 \leftarrow \epsilon_2$$

The value of lefthand operand ϵ_1 is interpreted as a pointer (address) which names a cell into which the value of the righthand operand, ϵ_2, is to be stored.*

Before turning to the issue of "why" the interpretation is as it is, I'd like to make three comments. First, the only people who have objected to the interpretation are those who first encounter it; to my

*The value of the store operator is ϵ_2, but that's not relevant to this discussion.

knowledge no one who is using the language objects. That only proves that it's possible to learn to live with it. Second, while the interpretation may be unique among higher level languages, it is precisely the interpretation adopted in assembly languages. Third, the interpretation is entirely consistent, the interpretation of an identifier is exactly the same independent of the context in which it occurs. (Maybe we could coin a phrase: "context-free semantics".)

Now, let me finally turn to why the interpretation is as it is. One of the fundamental design objectives of Bliss was to permit the user to define arbitrary representations of data structures by permitting him to define the accessing algorithm (expression) for elements of the structure. This implies not only that the user must be able to manipulate pointers as flexibly as values, but also that the value of an arbitrary expression must be able to stand as a name. This implies, for example, that the assignment operator must permit arbitrary expressions ϵ_1 and ϵ_2 in the context $\epsilon_1 \leftarrow \epsilon_2$.

An alternative to the Bliss interpretation of identifiers and dot operator is to assume that identifiers always represent the value of a variable and introduce another operator, say α, which means "the address of". One would still need the dot for several levels of indirection, but simple expressions such as (in current Bliss)

$$x \leftarrow .x+1$$

would be written

$$\alpha x \leftarrow x+1$$

Since, presumably, thre are fewer instances of addresses than values, there should be considerably fewer α's to write with this scheme than dots in current Bliss programs. Carrying this reasoning further, why

not presume α's on the left of assignments (or, almost, equivalently dots on the right)? Then one could write

$$x \leftarrow x+1$$

which is more familiar. Under this scheme one could, of course, write α's or (extra) dots to override the standard interpretation. Thus

$$.x \leftarrow 1$$

would store indirectly through x, and

$$x \leftarrow \alpha y$$

would store the address of y in x. Or would it? Let's examine some of the difficulties that arise from such an interpretation. None of these difficulties is insurmountable; however, they lead to a large collection of ad hoc interpretat on rules.

Above I suggested that $x \leftarrow \alpha y$ would store the address of y into x. One may think of α as either an operator, or merely as a compile time notation which overrides the suggested "value of" interpretation. If one chooses the first of these interpretations, then αy ought to mean the address of the value of y (i.e., $\alpha(.y)$)- which is not unique (there may be many locations whose current value is the same as that of y). Moreover, the expression $\alpha\epsilon$(where ϵ is an arbitrary expression) seems to have no useful interpretation unless one is willing to store ϵ, create a reference to this location, and support the garbage-collection that that implies. The "compile-time override" interpretation of α has its own set of problems; it makes 'αy' do something reasonable, but $\alpha\epsilon$ is nonsense and an arbitrary rule would have to be introduced to prohibit it. (What does $\alpha(1+2)$ mean?) On the other hand, $\alpha\epsilon$ is exactly what

you want in an expression such as

$$x \leftarrow \alpha y[i]$$

in which you wish to store the address of a structure element into x, so you must allow this case, too. It gets worse, as you'll see below.

Suppose, for the moment, that you've contrived some interpretation rules which handled the problems mentioned above, and that you move on to the implied α's (or dots). You are now faced with the problem of deciding what's on the left and what's on the right of an assignment operator. There's no problem with $x \leftarrow y$, but what about

$$(x+i) \leftarrow 5$$

Given the initial assumption that accessing is specified by an arbitrary algorithm, this is hardly an implausible thing to write. But what does it mean? It must be one of (in Bliss)

(a) $(x+i) \leftarrow 5$

(b) $(x+.i) \leftarrow 5$

(c) $(.x+i) \leftarrow 5$

(d) $(.x+.i) \leftarrow 5$

Relying on accumulated experience with respect to the usual way of storing vectors one might like for the interpretation to be (b), but I can find no rational reason for adopting this one; (a) or (d) seems more plausible, and (a) the most plausible. O.K., suppose you try to be consistent, and so you adopt (a) and then you write

$$(x+.i) \leftarrow 5$$

to explicitly indicate that, even though i appears on the left of an assignment, you want its value, not its address. You're now in trouble with another design objective of Bliss; namely, that the same accessing

function be usable everywhere. If you write

```
y←(x+.i)
```

which means (in Bliss)

```
y ← (.x+..i)
```

you do not get what was intended at all.

Again, you can gin-up a rule to cover this case. However, suppose that an accessing algorithm is specified by a function, f, and the body of f contains the expression "return x". Should this expression return the value or the address of x? In the expression

```
f()← f()+1
```

both are needed. Of course f could return both, but then consider

```
g()← g()+1
```

where the body of the routine g contains

```
return f()
```

Must g now return (1) the address of the address of x, (2) the address of the value of x, (3) the value of the address of x, and (4) the value of the value. WOW!

Having examined the consequences of some of the alternative proposals, let's now consider the reasons behind them. There are two: you are forced to write a lot of dots, and it deviates from the "standard", or "conventional". The first of these arguments has merit, and in fact was the rationale for choosing an inconspicuous, easily written and typed graphic for the "contents of" operator. In practice, however, users of the language have found little difficulty in either reading or writing the dot. The second argument is simply absurd. There is no standard since there are no other languages which deal with the same issues, except possibly assembly language, and Bliss uses the same convention as

assembly languages.

As for the virtues of the convention, it is simple and completely consistent, it permits accessing algorithms to be written and used in all contexts, and it covers all the cases. The distinction between name and value is a fundamental one, and in my opinion it is far more important to treat it explicitly and consistently than to provide minor convenience to the uninitiated.

Efficient Data Accessing in the Programming Language BLISS

David S. Wile and C. M. Geschke

EFFICIENT DATA ACCESSING IN THE PROGRAMMING LANGUAGE BLISS

David S. Wile and C. M. Geschke

Department of Computer Science

Carnegie-Mellon University

ABSTRACT

The specification of data structure in higher-level languages is isolated from the related specifications of data allocation and data type. Structure specification is claimed to be the definition of the accessing (addressing) function for items having the structure. Conventional techniques for data structure isolation in higher-level languages are examined and are found to suffer from a lack of clarity and efficiency.

The means by which data structure accessors may be defined in Bliss, the specification of their association with named, allocated storage, and their automatic invocation by reference to the named storage only, are discussed. An example is presented which illustrates their efficient implementation and their utility for separating the activities of data structure programming and algorithmic programming.

INTRODUCTION

Since the management and representation of data are of prime interest in programming, we wish to present the view of data structures that has been adopted in the implementation language Bliss. Bliss [1] is a higher-level language designed for writing large software systems for the PDP-10 [2] and is currently being implemented at Carnegie-Mellon University. Our paper is divided into two parts. First we discuss the issues which arise in defining and implementing data structures in higher-level languages. Then we present the facilities in Bliss which are designed to handle the representation of data.

HIGHER-LEVEL LANGUAGE DATA STRUCTURE SPECIFICATION

THREE ASPECTS OF DATA SPECIFICATION

We begin by considering three aspects of data structures which are not separable in most higher-level languages, but which can be separated in Bliss to allow greater flexibility in data specification:

1. Type specification - the name of a piece of data specifies its internal format and the class of operators for which it is a valid operand.
2. Allocation - the presence of a named data item requires that we be able to associate this name with its value; presumably, that value will require space in the underlying logical machine. The format (and perhaps the size) of the allocated space depends on the data type specified for the name. The scope rules of a language define the domain of valid access to a value via its name. The logical machine manages the allocation of space for storing the value and is free to overlay non-contemporaneous allocations.
3. Structure - the ability to structure regions of storage allows us to generate in a simple way a large collection of names and to retain the logical clarity of a generic name. Indeed we want the ability to compute a name (e.g., array subscript computation) and to sequence through a collection of names.

Taking Algol [3] as an example, the text

```
procedure P(A,B); real array B[1:100]; ...
```

provides a structure for B and types the elements of the structure (named: B[1], B[2], ..., B[100]). Furthermore, in addition to structuring and typing,

```
begin real array B[1:100]; ...
```

also allocates space. We emphasize: two different Algol implementations may physically structure the same logical structure differently (e.g., dope vector vs. by column or row).

IMPLEMENTING A FOREIGN DATA STRUCTURE

We consider in some detail how we build a data structure in a higher-level language whose inherent data structures may be quite different from those to be implemented. In particular consider a partial implementation of Lisp [4] in Algol. Atoms will be stored in an array with negative indices for non-null atoms and the zero index will indicate NIL. Cells will be stored in a two dimensional integer array with positive indices.

Now we examine two ways of implementing the Lisp accessing functions CAR and CDR.

```
(1)  integer array ATOMSPACE [-1000:0];
     integer array CELLSPACE [1:10000,1:2];
     integer procedure CAR(I); integer I;
         CAR := CELLSPACE[I,1];
     integer procedure CDR(I); integer I;
         CDR := CELLSPACE[I,2];

(2)  integer array ATOMSPACE [-1000:0];
     integer array CAR [1:10000], CDR[1:10000];
```

Note that in both implementations the Algol array bounds checking will handle the error resulting from attempting to access the CAR or CDR of an atom.

Several things are to be noted about these two implementations. Both (1) and (2) implement the same logical structure. The accessing structure is logically independent of the allocation since the declarations could appear in any Algol block at any level. The foreign types atom and pointer had to be incorporated into the structure of the implementing language. Implementation (1) has an advantage over (2) in that it can be modified more easily. We can change the body of the accessing functions CAR and CDR without changing the program's reference to them. On the other hand (2) is clearly more efficient than (1) since it employs the built-in accessing mechanisms of the Algol machine whereas (1) requires execution of the expensive procedure calling mechanisms of Algol procedures. Of course, neither implementation is as efficient as a direct machine language implementation of Lisp. Hence we can isolate a major difficulty that arises from specifying a data structure in a higher level language. In general we pay a high price in lost efficiency by implementing a data structure in a higher-level language unless, of course, that language is designed to make such implementations efficient. For example, if pointer or address were an Algol type, we could probably improve the above implementation to a point where the cost would be tolerable.

ISOLATING DATA ACCESS

We examine the motivation for isolating access to data. Consider the following Algol statement:

```
X := (Y[I] mod 2 ↑ (WORDLENGTH - 14) ÷ (2↑(WORDLENGTH - 22)); .
```

The code extracts bits 14 through 22 of Y[I] and stores it into X (where "WORDLENGTH" is the number of bits in a machine word and bits are numbered from the left). It seems evident that we would not want to write this rather cumbersome piece of code for each access of this subfield Y[I]. A major consideration in having structured identifiers in a language is to improve the clarity and readability of the program. It is also true that most programs are subject to fairly substantial modification as they are being built. Quite obviously the decision to change the format of the variable Y[I] so that the subfield of interest was no longer bits 14 through 22 but 7 through 15 would mean a laborious change of all the code that accessed that information.

At present most higher-level languages allow at best two ways of isolating accesses to data items whose structures are not built into the language--

macros and procedures. We can define one procedure as an accessor for a whole class of data items by passing information via parameters. Alternatively we can define a procedure as an accessor for a particular data item by allocating space for the data as an own variable of the procedure.

For example, assume that a linear array is being used to represent the elements of a symmetric matrix. The symmetry of the array allows the overlay of elements off the main diagonal. We define the following procedures for reading from and writing into arrays of this form:

```
real procedure LOADSYMMETRIC(A,I,J); real array A[1:100];
    integer I,J;
        LOADSYMMETRIC := if I > J
                         then A[I*(I-1)÷2+J]
                         else A[J*(J-1)÷2+I];
procedure STORESYMMETRIC(A,I,J,V); real array A[1:100];
    real V; integer I,J;
        if I > J
            then A[I*(I-1)÷2+J] := V
            else A[J*(J-1)÷2+I] := V;
```

The intention is for these accessing procedures to serve for several such arrays. If we wish to apply this structure to only one symmetric array, then the formal parameter A can be omitted (and A declared an own variable within the procedure).

We can avoid the expense of the function call mechanism by using string replacement macros.

```
macro LOADSYMMETRIC(A,I,J) =
    if I > J
        then A[I*(I-1)÷2+J]
        else A[J*(J-1)÷2+I];
macro STORESYMMETRIC (A,I,J,V)
    if I > J
      then A[I*(I-1)÷2+J] := V
      else A[J*(J-1)÷2+I] := V;
```

Both these solutions have drawbacks:

(a) As mentioned previously, function calls are unattractive because of their inefficiency.

(b) The presence of two accessing functions for one logical structure is required because of the left/right distinction in assignment statements.

(c) If a macro or procedure is defined for a whole class of data items and we decide to change the logical structure of one of the data items, then we must search the entire program for calls on the macro or procedure to change its structure.

(d) Macros have their own problems. Consider:

```
macro A(B,C) = if GLOBALBOOLEAN then B[C+3] else B[C-3]; .
```

If "GLOBALBOOLEAN is redeclared in an inner block, subsequent use of the macro will have the possibly undesirable effect of testing the new variable. Another unpleasant feature of the macro is the handling of actual parameters. Consider the macro call:

```
Y := LOADSYMMETRIC (X,F(I),G(J)); .
```

The expansion of this call produces inefficient and potentially side-effect-producing results because of the multiple calls on the functions F and G.

Having pointed out some of the issues that arise when considering how to implement data structures and having considered several of the problems associated with implementing data structures in higher-level languages, we next discuss how Bliss enables the programmer to specify his data structures and still maintain efficiency.

BLISS DATA STRUCTURE SPECIFICATION

NOTES ON BLISS

Bliss is primarily an Algol-like expression language with additional control expressions to circumvent problems encountered removing the "go to", and with declarations (for allocation) to facilitate independently compiled modules and special machine features (e.g., registers). The only anomaly which is relevant to this discussion is that names stand for machine addresses. If we want the contents of a named location, we must use a contents operator (the "."); e.g.,

y ← x+1;	adds 1 to the address of x and deposits it in the word addressed by y
(x+1)↚y;	deposits the contents of y into the word 1 past the address of x.

The PDP-10 has three types of data: instructions, addresses, and 36-bit words upon which machine operations may act. These types are determined dynamically by the interpreting hardware, and type checking is of a negative nature (e.g., "this is not a valid address"). The necessary inclusion of address manipulation facilities in any system implementation language would entail dynamic type checking if the logical type "address" were included. Visions of inefficiency thus lead to the inclusion of a single data type in Bliss: the 36-bit word. All operations are valid on this single data type.

Data allocation is by words in the machine; although fields within a word are addressable, there is no effective way of allocating a part of a word. Again, for efficiency reasons, Bliss allocates storage to programs in contiguous words. Allocation is done via explicit allocation declarations; a specified form of allocation is made, and the declared name is bound to the machine address of the beginning of the allocated storage. For example,

```
own A [200] ;
```

reserves 200 words of core (static) and binds the name "A" to the address of the allocation. The other static allocation declaration is for global storage. The effect of the allocation is the same as for owns, but the name becomes available to independently compiled modules which reference the variable via an external declaration.

Local variables are local to the block in which they are declared. They are allocated dynamically from the normal Algol implementation run-time-stack. The local variable name is dynamically bound to an address;

```
begin local Q, R [30] ; . . . end
```

allocates one word for Q, 30 for R and binds the names Q and R dynamically to their respective stack addresses. Recursive entry to a block causes recursive local allocation, unlike the own form. (This is simply the default form of allocation for Algol declarations; e.g. integer A, ...) The register allocation declaration requires compile time binding of addresses, but causes a recursive saving mechanism to be invoked; e.g.

```
begin register R1;  . . . end
```

causes the contents of the compile-time bound register named "R1" to be saved in the stack (and thereafter upon recursive entry to the block) and restored upon exit.

BLISS STRUCTURES

There are no structures "built-in" to Bliss as the array structure in Algol or the cell in Lisp. However, address arithmetic allows the use of any of the standard structures. For example, we can store the contents of C[.i, .j] into y (where C is a 7 x 9 array) by writing:

```
y←.(C+.i*9+.j);
```

(where we have presumed zero-origin indexing in both arguments and contiguous row storage allocation).

STRUCTURE DECLARATION--SIMPLE CASE

Naturally, expressions of the above form are quite common and their programming would become quite tedious without the structure declaration. Its form is easiest illustrated by example of a 7x9 array:

```
own C[63];
structure rowof9array[i,j] = .rowof9array+.i*9+.j;
map rowof9array C;
```

The first declaration allocates 7 * 9 = 63 words of core and binds the address of the allocation to the name "C". The structure declaration defines an "accessing template" for those names onto which it is mapped; its format is similar to that of a routine (procedure, function) declaration in which the body may reference the name of the structure as a formal parameter. The map declaration associates the structure "rowof9array" with the name "C". Thereafter, whenever the name "C" is used followed by a bracketed list of expressions, the effect is as if the structure were called as a routine with "C" as the actual corresponding to the routine name (which is used as a formal in the body) and the expressions as the actuals corresponding to the formals of the structure. Consider the routine declaration below:

```
routine  rrowof9array(rowof9array,i,j) = .rowof9array+.i*9+.j;
```

The effect of the use of C [3,5] in a program would then be the same as if we had (declared and) called rrowof9array(C,3,5). A Bliss routine is analagous to a valued procedure in Algol; however, the value of the routine is the value of the expression which is the body of the routine. A routine returns a 36-bit word, and hence, the returned value of a routine may be stored into.

```
rrowof9array(C,3,5)← 4
```

assigns the value 4 to array element C[3,5]. Remembering that C (without the dot) is an address, it should be clear that the above <u>effect</u> is the desired one.

Note that the Bliss contents operator removes the left/right-side distinction between structure accessing for storing and accessing for retrieval (drawback (b) above). Also, macro side-effects are not introduced (drawback (d)), for the structure is effectively equivalent to a routine, i.e., actual parameters are evaluated only once and identifiers in the structure body remain in the context of the structure declaration site.

However, we have introduced some additional drawbacks (soon to be removed):

(e) Although we have allowed the flexibility of choosing the accessing method, we must now write a different structure definition for each length row we have; e.g., rowof12array, or rowof7array.

(f) To allocate storage for the array C, the <u>own</u> above simply allocates the number in brackets of contiguous words--we must in some sense know how the structure works. Hence, in the above we had to know to allocate 7*9=63 words.

STRUCTURES AND MAPPING DECLARATIONS

Both (e) and (f) are solved in Algol by the array declaration:

"<u>integer</u> <u>array</u> C[1:7,1:9];".

Via the above, an Algol compiler knows to substitute 9 for the row length in the accessing expression and to allocate 7*9 words of core for the array.

Bliss extends the structure mechanism to facilitate this by the use of "incarnation formals". Use of the incarnation formals to a structure is indicated by not "dotting" the formal to a structure; e.g., in

```
structure array2[i,j] = .array2+.i*j+.j;
```

the first occurrence of j in the body refers to the incarnation formal. It is bound to the corresponding "incarnation actual" when the variable is mapped: e.g., <u>map</u> array2 C[7,9]; (in this case, 9).

Hence, the structure and routine correspondence:

```
structure array2[i,j] = .array2+.i*j+.j;
routine rarray2(incformali,incformalj,array2,i,j) =
        .array2+.i*.incformalj+.j;
```

applies, with the accessing expression for C[3,5] (in this case) having the <u>effect</u> of the routine call rarray2(7,9,C,3,5).

The structure writer knows best the allocation size required for variables onto which his structure will be mapped; hence, the "size expression" and "mapping declarations" were introduced into Bliss. The size expression is specified along with the structure declaration (preceding it, enclosed in brackets) as a function of the incarnation formals for the structure and of compile-time constants. All allocating declarations allow the mapping of a structure along with its declaration;

e.g.,
```
structure array2[i,j] = [i*j] .array2+.i*j+.j;
own array2 C[7,9];
```

The structure declaration defines a size expression, " [i*j] ", and accessing template, ".array2+.i*j+.j". The own declaration:

1. Maps "array2" onto "C";
2. Binds incarnation actual 7 to the incarnation formal i, and 9 to j;
3. Evaluates the size expression associated with the mapped structure with the incarnation actuals substituted; i.e. 7 * 9;
4. Allocates the number of words returned as the value of the size expression; i.e. 63;
5. Binds the name "C" to the address returned by the own allocation mechanism.

AN EXAMPLE

The utility of the Bliss data structure mechanism is illustrated by considering a solution to the following problem:

We wish to solve systems of linear equations with normalized upper-triangular coefficient matrices; i.e.,

$$(1)\quad x_i + \sum_{j=i+1}^{n} c_{ij}\, x_j = b_i \qquad \text{for } i = 1,2,\ldots,n$$

We must read the coefficient matrix and then solve the system for several sets of constraints. We also know we will be using a paged machine and that the coefficient matrices may be large.

Noting:

$$(a)\quad x_n = b_n$$

$$(b)\quad x_i + \sum_{j=i+1}^{n-1} c_{ij}\, x_j = b_i - c_{in}\, b_n \overset{\text{def}}{=} b_i' \quad \text{for } i=1,\ldots,n-1$$

(b) is a problem with the same specifications as (1) in one less variable. Thus, a solution technique is to iteratively subtract the product of the last found x_k with the column vector $(c_{1k}\ c_{2k}\ \ldots\ c_{k-1,k})$ from the (modified) constant vector $(b_1'\ b_2'\ \ldots\ b_{k-1}')$. This then becomes b' for the next step; i.e., new $b' = (b_1' - b_k' c_{1k}\ b_2' - b_k' c_{2k}\ \ldots\ b_{k-1}' - b_k' c_{k-1,k})$.

The algorithm portion (exluding I/O and declarations) in Algol might be:

```
for k := n step -1 until 2 do
   for i := k-1 step -1 until 1 do
      B[i] := B[i] - C[i,k] * B[k];
```

The solution is left in the original constant vector, B.

A Bliss implementation (with data structures and storage allocation specified) which mirrors the Algol program above is:

```
begin
   structure vector[i] = .vector+.i - 1;
   structure array2[i,j] = [i*j] .array2+(.i-1)*j+(.j-1);

   own vector B[n],
       array2 C[n,n];
                    .
                    .
                    .
                 INPUT
                    .
   decr k from n to 2 do
      decr i from .k-1 to 1 do
         B[.i] ← .B[.i] - .C[.i, .k]* .B[.k];
                    .
              OUTPUT & LOOP
                    .
                    .
                    .
end;
```

Now, note that the above solution:

(a) wastes space for the known zero and unity elements of the coefficient matrix;

(b) thrashes considerably (if n is large) in a paged machine, for the coefficient matrix is accessed by columns in decreasing index order, but is stored by row in increasing index order.

It can be seen that replacing the array2 structure with:

```
structure upperdiag[i,j] = [i*(i-1)/2].upperdiag + (.j-1)*(.j-2)/2 + .i-1;
```

and changing the mapping of "C" from "array2" to "upperdiag", modifies the program in such a way that it wastes no space for the known constant elements of "C" and eliminates thrashing by accessing elements in the same column of the coefficient matrix contiguously. The logical storage map of figure 1 may help to see this:

1,2	1,3	2,3	1,4	2,4	3,4	. . .	n-2,n	n-1,n

Figure 1.

The above change preserved the "algorithm portion" of the program--it continues to appear much the same as the Algol algorithm--however, the increase in overall efficiency is significant (presuming, for the moment, that the structure mechanism is efficient). The simplicity with which the change was accomplished indicates that "drawback (c)" has been removed.

SUBSTRUCTURES

Continuing to postpone the efficiency drawback, note that we would like to use a substructure on the columns of the coefficient matrix. We know that within the inner loop, each of the elements is taken from the same column, and thus the same multiplication ((.j-1)*(.j-2)/2) is repeated for each element in the column. We can indicate this substructure in Bliss via the bind declaration. This declaration is dynamic in the sense that the expression bound to is evaluated at execution time, upon entry of the block in which the bind occurs. For example, in

```
bind x=.y;
```

wherever x occurs in the block in which it is declared, the value of the contents of y will be considered its address.

The bind declaration allows its symbol to be mapped in a manner similar to the allocating declarations. Hence, we may write:

```
bind  array2  X [7,9] = .y+3;
```

This indicates to the compiler that the name "X" stands for the address which is the contents of y plus 3. If "X" is used as a structure access in the block in which the bind occurs, this address is to be considered the base of a 2-dimensional array with at most 7 rows and 9 columns (the semantics of the "array2" structure defined above).

Binding the name "COLUMNK" to the base of the kth column of "C" in the outer loop in the above program, we produce the more efficient and slightly more intuitive program:

```
begin
    % structure declarations for B and C %

    structure vector[i]= .vector+.i-1;
    structure upperdiag[i,j]=[i*(i-1)/2]
                              .upperdiag+(.j-2)*(.j-1)/2+.i-1;

    global vector B[n],
           upperdiag C[n,n];

    % Here we would begin the outer loop to read the
      coefficient matrix, "C".

      Here we would begin the inner loop to read the
      constant vector, "B". %

    decr k from n to 2 do
        begin
            bind vector COLUMNK=C[1,.k];
            decr i from .k-1 to 1 do
                B[.i]← .B[.i]-.COLUMNK[.i]*.B[.k];
        end

    % Here we would output or save the solutions which have
      been left in "B". Then we would continue the inner
      and outer loops. %

end;
```

EFFICIENCY

Clearly, the efficiency of structure accessing mechanisms highly affects their utility in a language which is designed for efficient implementation, A brief note about the compiler is necessary. The compiler first breaks program text into "lexemes"--atomic symbols for operators, reserved words, and identifiers. The lexeme for an identifier is unique within its scope; hence,

```
begin own b; begin own b; ... end; end;
```

causes the creation of two different lexemes for "b".

A structure access may best be understood as a lexeme-stream macro substitution mechanism,* where the structure body defines the lexeme-stream (with dots preceding formals removed). At a structure access, the actual parameters are evaluated (code is produced for their evaluation) and the incarnation actuals are retrieved. The compiler input is then taken from the structure lexeme-stream with actuals substituted.

Thus, under the array2 structure above,

```
C[2,1] ← .C[3,5] + 8
```

will compile as if we had written

```
(C+2*9+1) ← .(C+3*9+5) + 8
```

which, because of compiler optimization will compile as if we had written

```
(C+19) ← .(C+32) + 8
```

which will generate three machine instructions! The code compiled for our example is included as an appendix.

CONCLUSION

Bliss factors the separate issues of allocating storage, binding names to addresses and structuring the storage referenced by a name. Although all allocating declarations also bind names to the referenced store, names may be bound to addresses dynamically via the bind declaration which presumes the storage has been allocated for the contents of the named storage. A name may be structured using the map declaration independent of its allocation and binding. Because relationships often do exist between these three aspects of data structuring--allocating, binding and mapping--communication is allowed via "incarnation actuals", "size expressions" and "incarnation formals".

Use of the mapping, allocating declarations in Bliss permits the ease of use of other higher-level language declarations; the factoring of the issues of allocation, binding and structuring helps to separate the activities of data structure programming and algorithmic programming, while maintaining or, in fact, improving program efficiency.

---

*Structures are sometimes more efficiently accessed as routines. The current (unsatisfactory) solution is to compile those structures with declarations (other than their formal parameters) as routines.

ACKNOWLEDGMENTS

The concept of the structure declaration in Bliss is due to W. A. Wulf[1], who along with A. N. Habermann[1] and D. B. Russell[2] designed Bliss. We also with to thank our co-implementors, J. Apperson[1] and R. Brender[3].

REFERENCES

1. Wulf, W. A., Russell, D., Habermann, A. N., Geschke, C., Apperson, J., and Wile, D., Bliss Reference Manual, Department of Computer Science document, Carnegie-Mellon University, Pittsburgh, Pa., 1970.

2. Digital Equipment Corporation, PDP-10 Reference Handbook, 1969.

3. Naur, P. (Ed.), "Revised Report on the Algorithmic Language ALGOL 60", CACM 6, No. 1, (1963), pp. 1-17.

4. McCarthy, John, et.al., LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, 1962.

[1]Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa.

[2]Atlas Laboratory, Chilton, Didcot, Berks, England

[3]Digital Equipment Corporation, Maynard, Mass.

APPENDIX

```
0001    BEGIN
0002
0003      % MUST BIND AN UPPER BOUND FOR MATRIX DIMENSION %
0004
0005      BIND N=150;
0006
0007      % STRUCTURE DECLARATIONS FOR B AND C %
0010
0011      STRUCTURE VECTOR[I]= .VECTOR+.I-1;
0012      STRUCTURE UPPERDIAG[I,J]=[I*(I-1)/2]
0013                                .UPPERDIAG+(.J-1)*(.J-2)/2+/I-1;
0014
0015      GLOBAL THISN,
0016             VECTOR B[N],
0017             UPPERDIAG C[N,N];
0020
0021      % HERE WE WOULD BEGIN THE OUTER LOOP TO READ "THISN" (THE
0022        SIZE OF THIS ARRAY) AND THE COEFFICIENT MATRIX, "C".
0023
0024        HERE WE WOULD BEGIN THE INNER LOOP TO READ THE CONSTANT
0025        VECTOR, "B". %
0026
0027      DECR K FROM .THISN TO 2 DO
0030        BEGIN
0031          BIND VECTOR COLUMNK=C[1,.K];
0032          DECR I FROM .K-1 TO 1 DO
0033             B[.I]←.B[.I]-.COLUMNK[.I]*.B[.K];
0034        END;
0035
0036      % HERE WE WOULD OUTPUT OR SAVE THE SOLUTIONS WHICH HAVE
0037        BEEN LEFT IN "B".  THEN WE WOULD CONTINUE THE INNER AND
0040        OUTER LOOPS. %
0041
0042    END;
```

*

```
LINE    OFFSET  LABEL   OPCODE  REGISTER,ADDRESS(INDEX REG)

        0000            MOVE    13,THISN
0030    0001    L453:   CAIGE   13,2
        0002            JRST    $S,L460
0031    0003            ADD     $S,[000001,,000001]
0032    0004            MOVE    04,13
        0005            SUBI    04,2
        0006            MOVE    05,13
        0007            SUBI    05,1
        0010            IMUL    04,5
        0011            ASH     04,-1
        0012            HRRZI   06,C-1(4)
        0013            MOVEM   06,1($F)                ;LOCAL
        0014            MOVE    14,13
        0015            SUBI    14,1
0033    0016    L630:   CAIGE   14,1
        0017            JRST    $S,L503
0034    0020            MOVE    07,14
        0021            ADD     07,1($F)                ;LOCAL
        0022            MOVE    10,B-1(13)
        0023            IMUL    10,-1(07)
        0024            SUB     10,B-1(14)
        0025            MOVNM   10,B-1(14)
        0026            SOJA    14,L630  ↑↑↑
        0027    L503:   SUB     $S,[000001,,000001]
0035    0030            SOJA    13,L453  ↑↑↑
        0031    L460:   SETZ    $V,0

MODULE LENGTH =26+1
COMPILATION COMPLETE
```

*

HELP.DOC

William A. Wulf

BLISS DEBUGGING SUPPORT

***** V E R S I O N T W O *****

WM, A, WULF
APRIL 23, 1971
MODIFIED 2 SEP 71 MG MANUGIAN

INTRODUCTION

DDT MAY BE USED TO DEBUG PROGRAMS WRITTEN IN BLISS, HOWEVER, THE USE OF DDT ALONE REQUIRES A FAIRLY DETAILED KNOWLEDGE OF THE RUN-TIME REPRESENTATION OF BLISS PROGRAMS (STRUCTURE OF THE STACK, ETC.) AND IS NOT ESPECIALLY CONVENIENT, IN PARTICULAR, DDT CANNOT EXPLOIT ANY SPECIAL INFORMATION ABOUT THE STRUCTURE OF THE OBJECT PROGRAM, THE SERIOUS BLISS PROGRAMMER IS WELL ADVISED TO LEARN THE BLISS RUN-TIME STRUCTURE -- NEVERTHELESS, THERE ARE STILL A NUMBER OF DEBUGGING AIDS WHICH DDT DOES NOT PROVIDE, IN ORDER TO IMPROVE THE SITUATION, A MODULE CALLED "HELP" HAS BEEN WRITTEN TO AUGMENT THE FACILITIES OF DDT. THIS MODULE MAY BE LOADED (ALONG WITH DDT) WITH ANY BLISS PROGRAM -- ALTHOUGH RECOMPILATION OF HELP IS NECESSARY IF THE USER IS NOT USING THE STANDARD BLISS SYSTEM REGISTERS. "HELP" IS WRITTEN IN BLISS AND THEREFORE THE FACILITIESRIBED BELO BELOW MAY BE CALLED DIRECTLY FROM THE USERS SOURCE PROGRAM EVEN THOUGH THEY ARE PRIMARILY INTENDED FOR USE FROM DDT.

HOW TO USE HELP

1. THE ROUTINE(S) TO BE LOADED WITH HELP MUST CONTAIN THE TIMER SWITCH IN THE MODULE HEAD AND BE COMPILED WITH THE /T SWITCH. WITHOUT /T THE TIMER SWITCH IS INGNORED DURING COMPILATION AND, THEREFORE, MAY BE A PERMANENT PART OF A MODULE HEAD WITH NO HARM.

2. THE HELP MODULE MUST NOT BE COMPILED WITH /T.

3. THE MODULES TO BE DEBUGGED MUST BE LOADED WITH DDT AND HELP SUCH THAT DDT IS LOADED JUST ABOVE JOBDAT IN THE LOW SEGMENT. FOR EXAMPLE:

```
.DEB FOO,HELP
```

WORKS JUST FINE.

4. NOTE THAT THE FIRST FOUR WORDS OF EVERY ROUTINE ARE DEBUGGING OVERHEAD AND THAT ACTUAL CODE FOR THE ROUTINE ITSELF STARTS AT THE FIFTH WORD. TO TRACE A CALL TO A PARTICULAR ROUTINE, A BREAKPOINT MUST BE INSERTED AFTER WORD 4 OTHERWISE THE NECCESSARY HOUSEKEEPING DONE BY THE FIRST FOUR WORDS OF THE ROUTINE WILL NOT HAVE BEEN COMPLETED AND THE STACK WILL NOT BE SET UP FOR PROPER TRACING. LIKEWISE THE (LAST-SIX)TH WORD TO THE (LAST-ONE)TH WORD OF EACH ROUTINE ARE DEBUGGING OVERHEAD AND BREAKPONTS INSERTED IN THIS AREA WILL GIVE UNPREDICTABLE RESULTS. NOTE THAT THERE ARE NO RESTRICTIONS

IN PLACING BREAKPOINTS IN ACTUAL CODE OUTSIDE OF THE DEBUGGING PROLOG AND EPILOG.

5. THE ROUTINE BPN IN HELP MUST BE MODIFIED FOR EACH NEW VERSION OF DDT SINCE IT LOOKS AT THE DDT OBJECT CODE TO DETERMINE THE NUMBER OF THE LAST BREAKPOINT. IT IS CURRENTLY COMPATIBLE WITH DDT (VERSION 32, EDIT 23).
THE ONLY SET OF DEBUGGING ROUTINES
WHICH REQUIRE BPN IS THE XAREA(X) SET, THE OTHERS FUNCTION INDEPENTLY OF BPN.
TO MODIFY BPN APPROPRIATELY, DO THE FOLLOWING:

A. DETERMINE THE VALUE OF THE SYMBOLS

```
BCOM3
B1ADR
```

IN UDDT BY LOADING DDT.REL FROM SYS AND TYPING THEIR VALUES:

```
.LOA %S SYS:DDT
LOADING

LOADER NK CORE

EXIT

.DD


BCOM3=NNNN      B1ADR=MMMM
```

B. INSERT THE TWO VALUES JUST TYPED INTO THE APPROPRIATE BINDS IN BPN IN THE SOURCE OF HELP.BLI.

C. RECOMPILE HELP.BLI

OF COURSE, YOU MAY ALSO PATCH THE STANDARD VERSION OF HELP WITH DDT AFTER LOADING DDT, HELP, AND THE MODULE(S) TO BE DEBUGGED. CONSULT AN EXPANDED(/M) LISTING OF HELP TO DETERMINE WHICH LOCATIONS IN CORE TO MODIFY.

FACILITIES

THE FEATURES CURRENTLY IMPLEMENTED ALLOW DISPLAY OF THE USER'S STACK, TRACING OF CALLS ON SPECIFIC ROUTINES, DISPLAY OF VARIABLES AND REGIONS, AND AN EXTENSION OF THE ALT-MODE-X ($X) FEATURE OF DDT. THESE FEATURES ARE PROVIDED BY A SET OF GLOBAL ROUTINES IN THE HELP MODULE; THESE ROUTINES ARE DESCRIBED IN DETAIL BELOW.

THERE ARE THREE WAYS IN WHICH ONE OF THE ROUTINES IN HELP MAY BE ENTERED: A DIRECT CALL FROM THE USERS PROGRAM, FROM A DDT CONDITIONAL BREAK-POINT, OR BY EXECUTING A "PUSHJ" WITH THE DDT ALT-MODE-X FEATURE. THE READER IS PRESUMED TO BE FAMILIAR WITH THESE FEATURES OF DDT.

CONSIDER AN EXAMPLE: "XSTAK" IS ONE OF THE ROUTINES PROVIDED -- ITS EFFECT IS TO PRINT A DISPLAY OF THE USERS STACK, SHOWING THE ROUTINES CALLED, WHERE THEY WERE CALLED FROM, THEIR ACTUAL PARAMETERS, AND THEIR LOCAL VARIABLES. THE FORMAT OF THIS DISPLAY WILL BE DESCRIBED BELOW. NOW, SUPPOSE YOU HAVE A ROUTINE NAMED "THUD" AND YOU SET A DDT BREAKPOINT BY TYPING:

```
THUD+2$B
```

AT SOME LATER TIME, WHEN YOUR PROGRAM IS RUNNING A CALL WILL BE MADE ON THUD, THE BREAKPOINT WILL OCCUR, AND DDT WILL TYPE:

```
$NB>>THUD+2
```

AT THIS POINT YOU MAY DISPLAY THE CURRENT STACK BY USING "XSTAK" AND ENTERING IT VIA THE $X FEATURE -- IE, TYPE*

```
PUSHJ SREG,XSTAK$X
```

(BE SURE TO USE THE PROPER VALUE FOR "SREG" -- NORMALLY IT'S Ø.) AFTER THE DISPLAY IS FINISHED YOU'LL BE BACK IN DDT AND MAY PROCEED VIA AN $P, OR DO WHATEVER ELSE SUITS YOUR FANCY.

AN ALTERNATIVE TO THE EXAMPLE ABOVE IS TO USE THE CONDITIONAL BREAKPOINT FEATURE OF DDT, FOR EXAMPLE, SUPPOSE YOU SET BREAKPOINT #1 AT THUD BY TYPING

```
THUD+2$1B
```

AND SET THE CONDITIONAL BREAKPOINT INSTRUCTION AT $1B+1 TO THE SAME OLD PUSHJ:

```
$1B+1/ XXXX  PUSHJ SREG,XSTAK
```

NOW, AS SOON AS THE CALL ON THUD IS MADE THE STACK WILL AUTOMATICALLY GET THE STACK DISPLAY -- THEN THE BREAKPOINT WILL OCCUR. THIS MODE OF USING HELP IS MORE USEFUL WITH SOME OF THE OTHER HELP ROUTINES TO BE DESCRIBED BELOW.

ALL OF THE GLOBAL ROUTINES IN HELP HAVE NAMES OF THE FORM:

```
        XZZZZ
OR      XZZZZC
OR      XZZZZB
OR      XZZZZP
```

THAT IS, THEY ALL START WITH THE LETTER "X" FOLLOWED BY A FOUR CHARACTER NEMONIC, FOLLOWED BY A BLANK, A "C", A "B", OR A "P". ROUTINES WITH A COMMON "ZZZZ" ALL PERFORM THE SAME FUNCTION; THE SUFFIX DETERMINES WHAT HAPPENS AFTER THE FUNCTION IS COMPLETE. IN PARTICULAR THE FOLLOWING TABLE SUMMERIZES THE MEANING OF THE VARIOUS SUFFIX LETTERS:

SUFFIX	MEANING
BLANK	-IF CALLED FROM A USER PROGRAM, SIMPLY RETURN AND PROCEED AS USUAL. -IF CALLED BY $X, RETURN TO DDT TO PERMIT USER TO DO HIS THING. -IF CALLED FROM COND. BREAKPOINT, TREAT AS A "B" SUFFIX (SEE BELOW).
C	-FOR CONDITIONAL BREAKPOINT ONLY. AFTER COMPLETING FUNCTION CAUSE DDT TO DECREASE ITS PROCEED COUNT AND POSSIBLY BREAK.
B	FOR CONDITIONAL BREAKPOINT ONLY, AFTER COMPLETING FUNCTION FORCE A BREAK.
P	FOR CONDITIONAL BREAKPOINT ONLY, AFTER COMPLETINF FUNCTION FORBE PROGRAM TO PROCEED (LIKE AN $P).

THE GLOBAL ROUTINES PROVIDED IN THIS RELEASE , AND THEIR FUNCTIONS, ARE SUMMERIZED IN THE FOLLOWING TABLE:

ROUTINE	FUNCTION
XSTAK XSTAKC XSTAKB XSTAKP	DISPLAY THE USERS STACK IN THE FORM: A (← B+13) 110,,0 210,,1356 110,,13 211702,,X+1 B (← C+25) C (← D+44) 111,,1 ETC.

THE NAMES IN THE LEFT COLUMN ARE THOSE OF THE VARIOUS ROUTINES CALLED, ON THE SAME LINE IN THE LOCATION FROM WHICH THE ROUTINE WAS CALLED, EG. "(← C+25)", AND THE ACTUAL PARAMETERS DISPLAYED IN HALF-WORD OCTAL FORMAT. ON THE LINES BELOW THE CALL ARE THE VALUES OF THE LOCAL VARIABLES OF THE ROUTINE, NOTE THAT THE ACTUALS AND LOCALS ARE INDICATED BY POSITION - NOT NAME, ALSO, BE CAREFUL - SOME LOCALS ARE AUTOMATICALLY GENERATED BY THE COMPILER -- SO THE POSITION MAY NOT EXACTLY CORRESPOND WITH ITS DECLARATION POSITION. THE LOCAL POSITIONS DO CORRESPOND WITH THOSE SHOWN IN THE "/M" LISTING GENERATED BY THE COMPILER. THE INITIAL ROUTINE EXECUTED HAS A NULL CALLER.

XCALL
XCALLC
XCALLB
XCALLP

THESE ROUTINES DISPLAY, IN A FORMAT LIKE THAT ABOVE THE MOST RECENT ROUTINE CALL, ONE USEFUL APPLICATION OF THESE ROUTINES IS THAT OF TRACING THE EXECUTION OF ONE PARTICULAR ROUTINE, BY PLACING A CONDITIONAL BREAKPOINT AT THE HEAD OF THE ROUTINE TO BE TRACED AND A "PUSHJ SREG,XCALLP" IN THE APPROPRIATE COND-BP LOCATION A TRACE OF THE ROUTINE WITH ITS ACTUALS WILL BE OBTAINED.

XAREA
XAREAC
XAREAB
XAREAP

THESE ROUTINES DISPLAY A NUMBER (CURRENTLY 8) OF CONTIGUOUS AREAS OF MEMORY IN HALF WORD OCTAL FORMAT.
THE AREAS TO BE DISPLAYED ARE DEFINED BY NINE TABLES CALLED XAREA0, XAREA1,...., XAREA8. EACH OF THESE TABLES IS EIGHT WORDS LONG - THE FORMAT OF EACH WORD IN THESE TABLES IS:

IF ONE OF THESE ROUTINES IS ENTERED FROM CONDITIONAL BREAKPOINT #N, THEN THEY WILL PRINT THE REGIONS DESCRIBED BY THE TABLE "XAREAN".

IF, FOR EXAMPLE, YOU WANT TO DISPLAY A FIVE-WORD REGION WHOSE BASE ADDRESS IS "GLOP" EVERY TIME THE ROUTINE "THUD" IS CALLED YOU MIGHT TYPE:

```
THUD+2$1B
$1B+1/  XXXX    PUSHJ SREG,XAREAP
XAREA1/ XXXX    5,,GLOP
```

THEN SIT BACK AND WATCH.

XALTX

THIS ROUTINE IS A GENERALIZATION OF THE "$X" FEATURE OF DDT IN THE SENSE THAT IT PROVIDES AN INTERFACE BETWEEN DDT AND ANY ROUTINE WRITTEN IN BLISS. IT WORRIES ABOUT ALL THE MESSY DETAILS OF SAVING REGISTERS, ETC., NECESSARY TO GET FROM DDT INTO A BLISS ROUTINE AND BACK AGAIN WITHOUT DESTROYING THINGS ALONG THE WAY. THE ADDRESS OF THE BLISS ROUTINE TO BE CALLED IS SPECIFIED BY THE CONTENTS OF ONE OF THE WORDS: XALTX0, XALTX1,...,XALTX8. IF XALTX IS CALLED FROM CONDITIONAL BREAKPOINT #N (N=0 IF DIRECT OR EXPLICIT "$X" CALL) THEN THE CONTENTS OF XALTXN WILL BE USED TO SPECIFY THE ROUTINE TO BE CALLED.

THE ROUTINE CALLED INDIRECTLY THROUGH XALTX IS EXPECTED TO RETURN A VALUE OF 0,1, OR 2 - THESE VALUES ARE INTERPRETED LIKE THE C,B,AND P SUFFIXS RESPECTIVELY.

CONCLUSION

THE FACILITIES DESCRIBED ABOVE ARE A PRELIMINARY SET WHICH WILL BE EXPANDED IN THE FUTURE. I HOPE, NAY EXPECT, USERS OF HELP TO SUGGEST ADDITIONAL AND/OR REVISED FEATURES.

HELP.BLI

William A. Wulf

```
MODULE HELP=
BEGIN


!                       HELPER
!                ***** V E R S I O N  T W O  *****


!  THIS MODULE CONTAINS SEVERAL ROUTINES WHICH MAY BE LOADED WITH
!  A NORMAL BLISS MODULE TO PROVIDE DEBUGGING SUPPORT -- IT
!  PRESUMES THAT 'DDT' IS ALSO LOADED!
!

EXTERNAL DDTEND,DDT,JOBSYM,JOBREL,JOBHRL;


BIND BUFFLENGTH=15;
OWN BUFF[BUFFLENGTH];
OWN PBUFF;                          ! POINTER INTO THE OUTPUT BUFF

MACRO EBUFF= (BUFF+BUFFLENGTH)$,
      BBUFF= BUFF<36,7>$,
      SAVREGS= REGISTER I;I=15;DO PUSH(SREG,0,I) WHILE(I=,I=1) GTR 0;$,
      RESREGS= I=-15;DO POP(SREG,#20,I)WHILE(I=,I+1)LSS 0;$,
      SUBRET(L,V)= IF .(L)<0,18>LSS(DDTEND AND#777777) THEN
                          (L)=,(L)+(V);$,
      ENTER= SAVREGS;$,
      LEAVE(LL,VV)= RESREGS; SUBRET(LL,VV); .VREG $;

MACHOP  PUSH=#261, POP=#262;

BIND  AREASZ=8, NUMAREAS=8;

STRUCTURE SAT[I,J] = [I*J](.SAT+.I*AREASZ+,J)<0,36>;
GLOBAL      XAREA0;XAREA1;XAREA2;XAREA3;
               XAREA4;XAREA5;XAREA6;XAREA7;
               XAREA8[AREASZ];
BIND SAT  XAREAS = XAREA0;


GLOBAL      XALTX0,XALTX1,XALTX2,XALTX3,
            XALTX4,XALTX5,XALTX6,XALTX7,
            XALTX8;




FORWARD FIXF;

GLOBAL ROUTINE TIMER=
        (FIXF(#140,.JOBREL<0,18>); FIXF(#400010,.JOBHRL<0,18>));

  !FIX UP THE STACK THE WAY HELP EXPECTS IT TO BE.
  !I.E., INSURE THAT THE FREG IS PROPERLY PUSHED ONTO THE
  !STACK AT THE BEGINING AND POPPED OFF AT THE END OF EVERY
  !ROUTINE AS IT WAS IN THE GOOD OLD DAYS.

ROUTINE FIXF(START,FINISH)=
```

```
        BEGIN

        BIND F=FREG<0,0>, S=SREG<0,0>;

        MACRO MACHWORD(OP,AC,AD)= OP^27+AC^23+AD$,

              PUSHSF=MACHWORD(#261,S,F)$,
              PUSHS12=MACHWORD(#261,0,#12)$,
              POPSF=MACHWORD(#262,S,F)$,
              HRRZFS=MACHWORD(#550,F,S)$,
              JRSTHPL2=MACHWORD(#254,0,.I+2)$,
              JRSTHPL4=MACHWORD(#254,0,.I+4)$,
              JRSTHPL5=MACHWORD(#254,0,.I+4)$,
               JRSTHPL6=MACHWORD(#254,0,.I+5)$;

        MACHOP CALLI=#047, JRST=#254;

        REGISTER R;

        BIND SETUWP=#36;

        R←0;                          !TURN OFF HIGH SEG WRITE PROTECT
        CALLI (R,SETUWP);
        JRST (4,0);                   !HALT ON SETUWP ERROR

        INCR I FROM .START TO .FINISH - 11 DO
          IF .(@I)<18,18> EQL #551^9+#12^5 AND @(@I+1) EQL PUSHS12
            THEN IF .(@I+4)<27,9> EQL #265 %JSP%
              THEN
                BEGIN
                  (@I)<0,36>←JRSTHPL4;
                  UNTIL .(@I)<18,18> EQL #561^9+#12^5 DO I←.I+1;
                  IF @(@I+1) EQL PUSHS12
                    THEN (@I-1)<0,36>←JRSTHPL6;
                END
              ELSE
                BEGIN
                  (@I)<0,36>←JRSTHPL2;
                  (@I+2)<0,36>←PUSHSF;
                  (@I+3)<0,36>←HRRZFS;
                  UNTIL .(@I)<18,18> EQL #561^9+#12^5 DO I←.I+1;
                  IF @(@I+1) EQL PUSHS12
                    THEN
                      BEGIN
                        (@I-1)<0,36>←JRSTHPL5;
                        (@I+4)<0,36>←POPSF;
                      END
                END
    END;


ROUTINE F50T6(X)=
    IF .X EQL 0 THEN 0 ELSE
    IF .X LEQ #12 THEN .X+#17 ELSE
    IF .X LEQ #44 THEN .X+#26 ELSE
    IF .X EQL #45 THEN #16 ELSE
    IF .X EQL #46 THEN #04 ELSE #05;
```

```
ROUTINE B50T6(X)=
    BEGIN REGISTER R;
    X←.X AND #377777777777; R←0;
    DECR I FROM 5 TO 0 DO
        (R←.R↑(-6); R<30,6>←F50T6(.X MOD #50); X←.X DIV #50);
    .R
    END;

ROUTINE BPN=
    ! THIS ROUTINE MUST BE CHANGED FOR EACH VERSION OF DDT --
    ! OR, BETTER YET, DDT SHOULD BE CHANGED TO MAKE THE MOST
    ! RECENT BREAK POINT NUMBER AVAILABLE.
    BEGIN BIND BCOM3=#1536, B1ADR=#3627;
    (((@BCOM3-1) AND #777777)-(B1ADR-3))/3
    END;

ROUTINE SDDTST(X)=
    BEGIN REGISTER R,N; OWN ZN,ZZ; ZZ←ZN←0;
    R←.JOBSYM+1; N←ZZ;
    WHILE (R←.R+#2000002) LSS 0 DO
        IF (@@R-.X) LEQ 0 THEN
            IF (@@R-@@N) GEQ 0 THEN N←.R;
    .N-1
    END;

ROUTINE DUMP=
    BEGIN MACHOP CALLI=#47; REGISTER R;
    R←BUFF; CALLI(R,#3); R←BUFFLENGTH;
    DO BUFF[.R]←0 WHILE (R←.R-1) GEQ 0;
    PBUFF←BBUFF
    END;

ROUTINE INITHELP= (BUFF←0; DUMP());

ROUTINE PUT(X)=
    IF .X NEQ 0 THEN
        BEGIN
        IF .PBUFF EQL 0 THEN INITHELP() ELSE
        IF .PBUFF GEQ EBUFF THEN DUMP();
        REPLACEI(PBUFF,.X)
        END;

ROUTINE PUTS(X)=
    WHILE .X NEQ 0 DO (PUT(.X<28,7>); X←.X+7);

ROUTINE CRLF= (PUT(#15); PUT(#12); DUMP());

ROUTINE TAB= PUT(#11);

ROUTINE PRINT6(X)=
    BEGIN LOCAL L;
    DECR I FROM 5 TO 0 DO
        (L←.X<30,6>; X←.X+6; IF .L NEQ 0 THEN PUT(.L+#40));
    END;

ROUTINE PRINT50(X)= PRINT6(B50T6(.X));

ROUTINE PMOC(X)=
    BEGIN LOCAL T; T←0;
    DECR I FROM 11 TO 1 DO
```

```
        IF .X<3*.I,3> NEQ 0 THEN EXITLOOP (T←.I);
    DECR I FROM .T TO 0 DO PUT("0"←.X<3*.I,3>);
    END;

ROUTINE PDISP(X,T)=
    IF .X<0,18> LSS (DDTEND AND #777777) THEN PMOC(.X<0,18>) ELSE
    BEGIN LOCAL L;
    L←SDDTST(.X<0,18>); PRINT50(@@L);
    IF .T AND (L←=@(@L+1)←.X<0,18>) GTR 0 THEN
        (PUT("←"); PMOC(.L));
    END;

ROUTINE SPN(N)= INCR I FROM 1 TO .N DO PUT(" ");

ROUTINE P2C= (PUT(","); PUT(","));

ROUTINE SP3= SPN(3);

ROUTINE PWD(X)= (PMOC(.X<18,18>); P2C(); PDISP(.X<0,18>,1));

ROUTINE PWD2(X)=IF .X GEQ 0 THEN PWD(.X) ELSE (PUT("-");PMOC(-.X));

ROUTINE PWO(X)= (PMOC(.X<18,18>); P2C(); PMOC(.X<0,18>));

ROUTINE PRG(BASE,F,T)=
    INCR I FROM .F TO .T DO
        BEGIN
        PMOC(.I); PUTS(": "); PWD2(@(.BASE+.I-1)); SPN(4);
        IF NOT .I THEN (CRLF(); TAB());
        END;

ROUTINE PRC(F,CALLED)=
    BEGIN LOCAL NP,LP,CALLER;
    CALLER←.(.F-1)<0,18>-1;
    NP← IF .(@(.F-1))<27,9> NEQ #274 THEN 0 ELSE
        IF .(@(.F-1)-2)<27,9> NEQ #261 THEN 0 ELSE
        .(@@(.F-1))<0,18>;
    LP← .F-1-.NP;
    PDISP(.CALLED,0); TAB(); PUTS("(←"); IF .CALLER NEQ -1
    THEN PDISP(.CALLER,1); PUT(")");
    TAB(); PRG(.LP,1,.NP);
    .CALLER<0,18>+.NP+18
    END;

ROUTINE PSTK=
    BEGIN LOCAL F,CALLED,VAL,LL,NL;
    VAL←.VREG; F←@@@FREG; NL←@.FREG-.F-2; LL←.F+1;
    CALLED←.(@(.F-1)-1)<0,18>; CRLF();
    UNTIL .CALLED<0,18> EQL #777777 DO
        BEGIN LL←.F+1; CRLF();
        CALLED←PRC(.F,.CALLED<0,18>); CRLF(); TAB(); PRG(.LL,1,.NL);
        NL←@F-@@F-.CALLED<18,18>-2;
        F←@@F;
        END;
    .VAL
    END;

ROUTINE PFRC=(LOCAL F; CRLF(); F←@@@FREG; PRC(.F,,(@(.F-1)-1)<0,18>));

ROUTINE PAREA=
```

```
    BEGIN LOCAL J,K,N,BN; BN←BPN(); CRLF();
    INCR I FROM Ø TO AREAS²-1 DO
        BEGIN BIND AREA=.XAREAS[.BN,.I]<Ø,18>;
        CRLF(); J←.XAREAS[.BN,.I]<18,18>; N←Ø;
        IF AREA NEG Ø THEN
        DO(CRLF(); PDISP(AREA[.N],1); PUT("/"); TAB(); PWD2(@AREA[.N]))
                WHILE (N←.N+1; J←.J-1) GTR Ø;
        END;
    END;

GLOBAL ROUTINE XSTAK(X)=(ENTER; PSTK(); LEAVE(X+1,1));
GLOBAL ROUTINE XSTAKC(X)=(ENTER; PSTK(); LEAVE(X+1,Ø));
GLOBAL ROUTINE XSTAKB(X)=(ENTER; PSTK(); LEAVE(X+1,1));
GLOBAL ROUTINE XSTAKP(X)=(ENTER; PSTK(); LEAVE(X+1,2));


GLOBAL ROUTINE XCALL(X)=(ENTER; PFRC(); LEAVE(X+1,1));
GLOBAL ROUTINE XCALLC(X)=(ENTER; PFRC(); LEAVE(X+1,Ø));
GLOBAL ROUTINE XCALLB(X)=(ENTER; PFRC(); LEAVE(X+1,1));
GLOBAL ROUTINE XCALLP(X)=(ENTER; PFRC(); LEAVE(X+1,2));


GLOBAL ROUTINE XAREA(X)=(ENTER; PAREA(); LEAVE(X+1,1));
GLOBAL ROUTINE XAREAC(X)=(ENTER; PAREA(); LEAVE(X+1,Ø));
GLOBAL ROUTINE XAREAB(X)=(ENTER; PAREA(); LEAVE(X+1,1));
GLOBAL ROUTINE XAREAP(X)=(ENTER; PAREA(); LEAVE(X+1,2));


GLOBAL ROUTINE XALTX(X)=(LOCAL L; ENTER; L←(@XALTXØ[BPN()])();
    LEAVE(X+1,.L));



END

ELUDOM
```

TIMER.DOC

Joseph M. Newcomer

00100 - - - - - B L I S S T I M E R M O D U L E - - - - -
00200
00300 THE BLISS TIMER MODULE CONSISTS OF A SET OF ROUTINES
00400 WHICH ENABLE THE USER TO GATHER TIMING STATISTICS ON BLISS
00500 PROGRAMS DURING EXECUTION. THE TIMING SYSTEM CONSISTS OF
00600 THE BLISS MODULE "TIMER" AND THE MACRO-10 MODULE "TIMINT".
00700
00800 IN ORDER FOR THE TIMING ROUTINES TO FUNCTION, THE TIMER
00900 ROUTINES AND THE SYSTEM TO BE MONITORED MUST BE LOADED WITH
01000 DDT (THE /D SWITCH TO THE LOADER, XD SWITCH TO CCL, OR THE
01100 DEBUG COMMAND ALL ACCOMPLISH THIS). ASSUMING THE USER
01200 WISHES TO MODIFY AND RECOMPILE HIS MAIN PROGRAM (A WAY TO
01300 AVOID THIS IS DISCUSSED BELOW), HE MUST ADD:
01400
01500 EXTERNAL TIMSET,TIMEND;
01600
01700 TO HIS DECLARATIONS, AND THE CALLS IN THIS MANNER:
01800
01900 < BEGINNING OF MAIN PROGRAM >
02000
02100 TIMSET();
02200
02300 < MAIN BODY OF MAIN PROGRAM >
02400
02500 TIMEND();
02600
02700 < END OF MAIN PROGRAM >
02800
02900
03000 THE <BEGINNING> MAY INCLUDE STACK INITIALIZATION IF NOT
03100 DONE IMPLICITLY BY THE STACK DECLARATION IN THE MODULE HEAD,
03200 PLUS ANY PROCESSING THE USER WISHES TO DO BEFORE TIMING
03300 BEGINS. ALL CODE EXECUTED BETWEEN TIMSET() AND TIMEND()
03400 WILL BE MONITORED. THE <END> MAY CONTAIN ANY OTHER
03500 PROCESSING. IN PARTICULAR, THE <END> MAY CONTAIN CALLS ON
03600 THE OUTPUT ROUTINES DISCUSSED BELOW.

```
00100    - - - - - - - - T I M I N G   O U T P U T - - - - - - - -
00200
00300
00400        THE  COLLECTED  STATISTICS  ARE  SORTED BY THE REPORTING
00500    ROUTINES AND OUTPUT IN THE FOLLOWING FORMAT:
00600
00700    LOST TIME ******** = ***%
00800
00900    METERED TIME ******** = ***%
01000
01100    TOTAL TIME ******** = 100%
01200
01300    OVERHEAD RATIO ****%
01400
01500    DEPTH OF CALLS  ******
01600
01700    STACK LEFT *******
01800
01900    TOTAL RTNS ********
02000
02100    TOTAL CALLS *********
02200
02300    NAME       ---CALLS--- ---ROUTINE---  -CUMULATIVE-  RTN AVG  CUM AVG
02400    AAAAAA  ******** ***% ******** ***%  ******* ***%  ******** ********
02500
02600
02700        "LOST TIME" IS THE PERCENTAGE OF  TOTAL  EXECUTION  TIME
02800    THAT  WAS  SPENT  ACCUMULATING STATISTICS,  THIS IS PROVIDED
02900    FOR INFORMATION ONLY; IT DOES NOT INDICATE THAT  THE  ACTUAL
03000    FIGURES  HAVE  AN  ERROR INTRODUCED BY THE FACT THE ROUTINES
03100    HAVE BEEN TIMED.
03200
03300        "METERED TIME" IS THE ACTUAL TIME SPENT IN EXECUTING THE
03400    USER'S CODE.
03500
03600        "TOTAL  TIME"  IS  THE SUM OF THE TWO ABOVE TIMES AND IS
03700    THE TOTAL EXECUTION TIME OF THE PROGRAM BEING MEASURED, FROM
03800    THE RETURN FROM TIMSET() TO THE CALL ON TIMEND().
03900
04000        "OVERHEAD  RATIO"  IS  THE PERCENTAGE BY WHICH EXECUTION
04100    TIME INCREASED AS A RESULT OF THE SYSTEM BEING  TIMED,  THIS
04200    IS THE COST OF MAKING THE MEASUREMENTS.
04300
04400        "DEPTH OF CALLS" IS THE MAXIMUM  DEPTH  TO  WHICH  CALLS
04500    WERE DYNAMICALLY NESTED.
04600
04700        "STACK   LEFT"   IS   THE   MINIMUM   NUMBER   OF  WORDS
04800    (APPROXIMATELY) LEFT AT THE TOP OF THE STACK AT THE  DEEPEST
04900    CALL.   TO  COMPUTE THE MAXIMUM DEPTH OF THE STACK, SUBTRACT
05000    THIS VALUE FROM YOUR STACK SIZE.
05100
05200        "TOTAL CALLS" IS THE TOTAL  NUMBER  OF  ROUTINE  ENTRIES
05300    PERFORMED.
05400
05500        THE  REMAINING FIGURES COME OUT TABULATED IN COLUMNS, AS
05600    FOLLOWS:
05700
05800
05900        THE "NAME" COLUMN CONTAINS THE NAME OF THE BLISS ROUTINE
06000    OR FUNCTION.
```

THE "CALLS" COLUMN CONTAINS TWO FIGURES: THE NUMBER OF TIMES THE ROUTINE WAS CALLED, AND THE PERCENTAGE OF THE TOTAL CALLS WHICH THIS CONSTITUTED.

THE "ROUTINE" COLUMN CONTAINS TWO FIGURES: THE TOTAL AMOUNT OF TIME SPENT IN THE ROUTINE, EXCLUSIVE OF ITS SUBROUTINES AND THE PERCENTAGE OF THE TOTAL METERED TIME WHICH THIS CONSTITUTED.

THE "CUMULATIVE" COLUMN CONTAINS TWO FIGURES: THE TOTAL AMOUNT OF TIME SPENT IN THE ROUTINE, INCLUDING ALL ITS SUBROUTINES (WHICH MAY INCLUDE ITSELF), AND THE PERCENTAGE OF TOTAL METERED TIME WHICH THIS CONSTITUTED.

THE "RTN AVG" TIME IS THE ROUTINE TIME DIVIDED BY THE NUMBER OF CALLS; THE "CUM AVG" TIME IS THE CUMULATIVE TIME DIVIDED BY THE NUMBER OF CALLS.

ALL TIMES GIVEN ARE IN "TICKS", WHERE A TICK IS 10 MICROSECONDS.

THE OUTPUT MAY BE SORTED IN ANY OF THE AVAILABLE FIGURES STORED BY THE TIMESORT() ROUTINE; HOWEVER, SEVERAL SORTS ARE PRE-SPECIFIED AND INCLUDE OUTPUTTING OF THE SORTED DATA. THESE ARE:

TIMST1 SORTED BY NAMES, ASCENDING.

TIMST2 SORTED BY TOTAL CALLS, DESCENDING.

TIMST3 SORTED BY ROUTINE TIMES, DESCENDING.

TIMST4 SORTED BY CUMULATIVE TIMES, DESCENDING.

TIMST5 SORTED BY AVERAGE ROUTINE TIME, DESCENDING.

TIMST6 SORTED BY AVERAGE CUMULATIVE TIMES, DESCENDING.

TIMST7 SORTED BY ADDRESSES, ASCENDING.

ALTHOUGH TIMST7 DOES NOT APPEAR OBVIOUSLY USEFUL, CONSIDER THE PROBLEM OF FINDING OUT WHICH MEMORY AREAS ARE MOST HIGHLY ACCESSED IN A SYSTEM RUNNING ON A PAGED MACHINE.

THESE ROUTINES MAY BE CALLED FROM THE USER'S MAIN PROGRAM BY DECLARING THEM "EXTERNAL", OR FROM DDT (SEE BELOW).

THE COLUMN ON WHICH THE OUTPUT IS SORTED IS INDICATED BY AN ASTERISK ABOVE THE COLUMN. NOTE THAT THE NUMBER OF THE SORT (1-6) CORRESPONDS TO THE COLUMN POSITION OF THE DATA SORTED.

IN ADDITION TO THESE REPORTING ROUTINES, A ROUTINE "TIMALL()" IS AVAILABLE WHICH CALLS ALL THE TIMING REPORTS (TIMST1-TIMST7) AS WELL AS THE LOCALIZATION REPORTS (TIMST8-TIMST9 [SEE BELOW]).

00100 - - - - - L I N E P R I N T E R - - - - -

00200
00300 OUTPUT MAY BE DIRECTED TO THE LINE PRINTER BY CALLING
00400 THE ROUTINE "TIMLPT()". ALL FURTHER OUTPUT WILL BE DIRECTED
00500 TO THE LINE PRINTER (LPT) UNTIL REDIRECTED TO THE TTY BY A
00600 CALL TO "TIMTTY()". TIMLPT() SHOULD NOT BE CALLED UNTIL
00700 AFTER TIMEND() IS CALLED. THESE TWO ROUTINES MAY ALSO BE
00800 CALLED FROM DDT.
00900
01000 ALL I/O IN THE USER'S PROGRAM SHOULD BE CORRECTLY
01100 TERMINATED, SINCE A "CALL [SIXBIT /RESET/]" UUO IS EXECUTED
01200 PRIOR TO EACH PRINTING. NOTE THIS ALSO RESETS JOBFF TO
01300 .JOBSA<18,18> AND SETS THE WRITE-PROTECT BIT IN THE HIGH
01400 SEGMENT. IF ANY OF THESE HAVE AN ADVERSE EFFECT ON THE
01500 PROGRAM OR DATA BASE, THEN PRINTER OUTPUT MAY NOT BE USED.
01600
01700 IF SPOOLING IS IN OPERATION, EACH SET OF STATISTICS IS A
01800 SEPARATE LISTING.
01900
02000 THE CHANNEL USED FOR LPT OUTPUT IS SPECIFIED BY A MACRO
02100 IN THE BEGINNING OF THE PROGRAM. IT MAY BE CHANGED BY THE
02200 USER (SEE: STANDARD MODIFICATIONS).
02300

- - - S T A N D A R D M O D I F I C A T I O N S - - -

THE TIMER MODULE CURRENTLY CONTAINS ITS OWN DATA AREAS AS "OWN" STRUCTURES, THE SIZES COMPILED INTO IT AT THE MOMENT MAY NOT BE SUITABLE FOR ALL SYSTEMS; THEY MAY BE TOO SMALL (OR EVEN TOO LARGE!). IF AN ERROR OCCURS WHILE RUNNING BECAUSE THESE AREAS ARE TOO SMALL, A MESSAGE WILL BE OUTPUT INDICATING WHICH TABLE OVERFLOWED. THE TWO PARAMETERS ARE MACROS IN THE FIRST FEW LINES OF SOURCE PROGRAM; "MAXRTN" IS THE NUMBER OF ROUTINES WHICH CAN BE TIMED (E.G., IT CURRENTLY IS SET TO "200"). IF MORE THAN "MAXRTN" ROUTINES ARE IN THE SYSTEM TO BE TIMED, A MESSAGE WILL BE PRINTED INDICATING HOW MANY ARE REQUIRED. THE VALUE OF "MAXRTN" MUST BE CHANGED, AND THE TIMER MODULE RECOMPILED. THE OTHER PARAMETER, "MAXDEEP", INDICATES THE DEPTH TO WHICH ROUTINES MAY BE DYNAMICALLY NESTED. A STACK IS USED TO KEEP TRACK OF NESTING, AND "MAXDEEP" DECLARES THE SIZE OF THIS STACK.

IF DECLARED REGISTERS OR RESERVED REGISTERS ARE USED BY THE SYSTEM BEING TIMED, THEN THE MODULE HEAD OF THE TIMER MODULE MUST BE ALTERED AND THE MODULE RECOMPILED.

IF THE SYSTEM BEING TESTED IS BEING LOADED WITH A HIGH SEGMENT ADDRESS OTHER THAN #400000, THE MACRO "HISEGAD" MUST BE CHANGED TO REFLECT THIS.

IF THE LOCALIZATION MEASURES ARE DESIRED FOR BLOCKS OF MEMORY OTHER THAN 1024 WORDS IN SIZE (NO LESS THAN THIS, HOWEVER), THE MACRO "COREBLOCK" MUST BE CHANGED TO REFLECT THIS. THE VALUE OF COREBLOCK IS N FOR A BLOCK OF SIZE 2**N (E.G., 1024= 2**10, SO COREBLOCK=10).

THE CHANNEL NUMBER USED FOR LPT OUTPUT IS #16. THIS IS TO PREVENT CONFLICTS WITH THE DDT PATCH FILE I/O WHICH USES CHANNEL NUMBER #17. IF IT IS DESIRED TO CHANGE THIS ASSIGNMENT, CHANGE THE MACRO "LPTCHNL" IN THE BEGINNING OF THE MODULE AND RECOMPILE.

00100 - - N O N S T A N D A R D M O D I F I C A T I O N S - -
00200
00300
00400 IF THE DEFAULT VALUES OF FREG, SREG, AND VREG ARE NOT
00500 USED, THEN THE MODULE HEAD OF THE TIMER MODULE MUST BE
00600 CHANGED TO REFLECT THIS, AND THE TIMER MODULE RECOMPILED.
00700 IN ADDITION, THE DECLARATIONS IN THE MACRO-10 MODULE
00800 "TIMINT" MUST BE CHANGED TO REFLECT THE NEW VALUES, AND THIS
00900 MODULE REASSEMBLED.
01000

00100 - - - - - - - - R E S T R I C T I O N S - - - - - - - -
00200
00300
00400 THE PROGRAM MAY NOT BE RESTARTED AFTER TIMSET() IS
00500 CALLED. THIS WILL BE FIXED WHENEVER THE LOADER BUG WHICH
00600 ACCIDENTLY OVERLAYS "OWN" DATA (INSTEAD OF LEAVING IT
00700 ZEROED) IS FIXED.
00800
00900 ANY ROUTINE WITH A NAME SIX (OR MORE) CHARACTERS IN
01000 LENGTH WHOSE FIRST THREE CHARACTERS ARE "TIM" WILL NOT BE
01100 TIMED EXPLICITLY. THIS TEST IS USED TO DIFFERENTIATE
01200 ROUTINES OF THE TIMING PACKAGE FROM THOSE OF THE USER.
01300 SHOULD THE USER HAVE ANY ROUTINES OF THIS NATURE, THE TIME
01400 SPENT IN THEM WILL BE CHARGED TO THEIR CALLER.
01500
01600 IF DDT IS USED TO START THE TIMING OFF (SEE BELOW),
01700 BREAKPOINTS MUST NOT BE PLACED AT ANY ROUTINE ENTRY POINTS
01800 BEFORE TIMSET() IS CALLED. IF ONE IS PLACED IN SUCH A
01900 POSITION, THE ROUTINE WILL NOT BE TIMED EXPLICITLY, BUT
02000 RATHER AS DESCRIBED ABOVE (FOR "TIMXXX" ROUTINES).
02100 ESPECIALLY ONE SHOULD NOT PLACE A BREAKPOINT AT THE POPJ
02200 WHICH LEAVES THE ROUTINE. THE SIDE EFFECTS THIS COULD HAVE
02300 ARE TOO HORRIFYING TO CONTEMPLATE.
02400
02500 THE HIGH SEGMENT USED MUST BE PRIVATE, SINCE THE
02600 TIMINIT() ROUTINE (CALLED BY TIMSET()) EXERCISES WRITE
02700 PRIVILEGES IN THE HIGH SEGMENT.
02800
02900 THE ROUTINES MUST NOT CONTAIN SPURIOUS POPJ INSTRUCTIONS
03000 (WHICH CAN BE GENERATED BY USE OF THE MACHOP FEATURE IN
03100 BLISS). ONE, AND ONLY ONE, POPJ IS PERMITTED IN A ROUTINE.
03200
03300 IF A MACRO-10 SUBPROGRAM IS USED, IT MUST ADHERE TO THE
03400 BLISS LINKAGE DISCIPLINES IF IT IS TO BE EXPLICITLY TIMED.
03500 IN PARTICULAR, IT MAY CONTAIN ONLY ONE "PUSH SREG,FREG"
03600 INSTRUCTION WITH A LABEL, EITHER INTERNAL, OR EXTERNAL.
03700 (NOTE THAT SUCH INSTRUCTIONS WITHOUT LABELS ATTATCHED ARE
03800 VALID). IT MUST ALSO CONTAIN ONE AND ONLY ONE POPJ
03900 INSTRUCTION (SEE ABOVE RESTRICTION). VIOLATION OF THIS RULE
04000 WILL RESULT IN ABSOLUTELY UNPREDICTABLE BUT CERTAINLY
04100 INCORRECT BEHAVIOR OF THE PROGRAM BEING TIMED.
04200
04300 THE LOCATION OF THE HIGH SEGMENT MUST NOT BE CHANGED AT
04400 RUN TIME WITH A CORE OR REMAP UUO IF ANY ROUTINES BEING
04500 TIMED ARE IN THE HIGH SEGMENT. THE CHANGE WILL NOT BE
04600 DETECTED BY THE TIMING PACKAGE AND CONFUSION AND CATASTROPHE
04700 WILL ENSUE. THE SIZE OF EITHER THE LOW OR HIGH SEGMENT MAY
04800 BE CHANGED, AS LONG AS THIS DOES NOT RESULT IN CHANGING THE
04900 ORIGIN OF THE HIGH SEGMENT.
05000
05100 IF OUTPUT IS TO BE DIRECTED TO THE LINE PRINTER, ALL I/O
05200 IN THE USER'S PROGRAM MUST BE CORRECTLY TERMINATED, SINCE A
05300 "CALL [SIXBIT /RESET/]" UUO IS EXECUTED PRIOR TO EACH
05400 PRINTING. NOTE THIS ALSO RESETS JOBFF TO .JOBSA<18,18> AND
05500 SETS THE WRITE-PROTECT BIT IN THE HIGH SEGMENT. IF ANY OF
05600 THESE HAVE AN ADVERSE EFFECT ON THE PROGRAM OR DATA BASE,
05700 THEN PRINTER OUTPUT MAY NOT BE USED.
05800

```
- - - - - - - - U S E   F R O M   D D T - - - - - - - -


    THE TIMING PACKAGE MAY BE CALLED FROM DDT (RATHER THAN
HAVING TO RE-ASSEMBLE THE MAIN PROGRAM MODULE) BY PLACING A
BREAKPOINT IN THE MAIN PROGRAM.  THIS BREAKPOINT MUST BE SET
SOMEPLACE AFTER THE STACK HAS BEEN INITIALIZED, BUT BEFORE
THE FIRST CALL ON A ROUTINE TO BE TIMED.  THE CALL TO TIMSET
FROM DDT MUST BE MADE FROM THE CONTEXT OF THE MAIN PROGRAM.
A BREAKPOINT MUST ALSO BE PLACED SOMEWHERE IN THE MAIN
PROGRAM WHERE TIMING IS TO CEASE.  A GOOD PLACE, FOR
EXAMPLE, IS THE "UUO 12" AT THE END OF THE CODE.

    WHEN THE FIRST BREAKPOINT IS REACHED, TYPE "PUSHJ
TIMSET$X".  THIS WILL CALL THE TIMSET() ROUTINE.  WHEN
CONTROL RETURNS, TYPE "$P" TO PROCEED.  WHEN THE SECOND
BREAKPOINT IS REACHED, TYPE "PUSHJ TIMEND$X" TO TERMINATE
TIMING.  THIS WILL MARGINALLY INFLUENCE THE TIMINGS OF THE
MAIN PROGRAM, SINCE THE TIME SPENT IN DDT AFTER THE RETURN
FROM TIMSET AND BEFORE THE CALL OF TIMEND ARE CHARGED TO THE
MAIN PROGRAM.

    AFTER CONTROL RETURNS FROM THE SECOND PUSHJ, TYPE "PUSHJ
TIMST#$X" (WHERE # IS ONE OF THE NUMBERS 1-9) TO OBTAIN
OUTPUT OF THE STATISTICS.  ALTERNATIVELY, ONE MIGHT TYPE
"PUSHJ TIMALL$X" TO OBTAIN OUTPUT OF ALL THE STATISTICS.

    TO DIRECT OUTPUT TO THE LINE PRINTER, TYPE "PUSHJ
TIMLPT$X".  ALL OUTPUT WILL BE DIRECTED TO THE LINE PRINTER
UNTIL REDIRECTED TO THE TTY BY "PUSHJ TIMTTY$X".  THESE
CALLS SHOULD NOT BE GIVEN UNTIL TIMEND() HAS BEEN CALLED.

    TO AID IN SETTING UP A PROGRAM TO BE TIMED, A "PATCH
FILE" MAY BE USED.  THIS CONTAINS ALL THE DDT COMMANDS
NECESSARY TO SET UP TIMING.  GENERALLY THESE CONSIST ONLY OF
SETTING UP BREAKPOINTS AND EXECUTING THE INITIALIZATION
COMMANDS, BUT MORE COMPLEX OPERATIONS MAY BE NECESSARY.  THE
PROTOCOL BELOW SHOWS HOW TO SET UP AND USE A PATCH FILE:


.MAKE PAT1.DDT
*@I\ DRIV.F+4$B DRIV.F+5$B  $G  PUSHJ TIMSET$X $P\$$
*EX$$
EXIT
↑C

.GET DSK TIMING
JOB SETUP
↑C

.DDT

$".PAT1.$Y      DRIV.F+4$B      DRIV.F+5$B      $G
$1B>>DRIV.F+4           PUSHJ TIMSET$X

$P


    NOTE THAT AFTER TYPING $",PAT1,$Y THE COMMANDS IN THE
FILE ARE TYPED OUT AS IF THEY HAD BEEN TYPED FROM THE TTY.
```

```
06100   THE SY ("YANK") COMMAND IS MORE FULLY DOCUMENTED IN THE C-MU
06200   DDT MODIFICATIONS WRITEUP.  IT IS RECOMMENDED THAT THE LAST
06300   COMMAND IN THE FILE BE $P, AND THAT THE CALLS TO "TIMEND"
06400   AND THE REPORTING ROUTINES BE PERFORMED FROM THE TTY.  THE
06500   MAIN REASON FOR THIS IS THAT IF DDT IS RE-ENTERED AT ANY
06600   POINT BEFORE THE DESIRED BREAKPOINT OCCURS, THE REMAINDER OF
06700   THE FILE WILL BE READ WITH UNDESIRABLE SIDE EFFECTS.
06800
```

```
00100   - - - M E T H O D   O F   I M P L E M E N T A T I O N - - -
00200
00300
00400       THE  TIMER IS READ BY USE OF A CALL AC,[SIXBIT /RUNTIM/]
00500   UUO, MODIFIED ACCORDING TO THE  MEMO  CIRCULATED  DESCRIBING
00600   THE  HIGH-RESOLUTION TIMER IMPLEMENTATION AT C-MU.  IN ORDER
00700   TO OPERATE CORRECTLY  ON  SYSTEMS  WHICH  DO  NOT  HAVE  THE
00800   HARDWARE  AND  SOFTWARE MODIFICATIONS FOR THIS, THE "TIMINT"
00900   PROGRAM WILL HAVE TO BE CHANGED TO THEIR SPECIFICATIONS.  AS
01000   A  WARNING,  THE  "JIFFY TIMER" OF THE STANDARD DEC SOFTWARE
01100   HAS TOO COARSE A RESOLUTION (1/60 OR 1/50 OF  A  SECOND)  TO
01200   MAKE  TIMING  SHORT ROUTINES POSSIBLE, AND ALSO SUFFERS FROM
01300   THE FACT THAT INTERRUPTS FROM DEVICES  GET  CHARGED  TO  THE
01400   RUNNING  JOB,  REGARDLESS  OF WHETHER THAT JOB GENERATED THE
01500   REQUEST OR NOT.
01600
01700       ALL TIMING FIGURES GIVEN ARE IN "TICKS",  WHICH  ARE  10
01800   MICROSECONDS   EACH,    HENCE    THE   35-BIT   INTEGER   WHICH
01900   REPRESENTS TIME CAN COUNT 34,359,738,368 TICKS,  OR  343,597
02000   SECONDS, MORE THAN ADEQUATE FOR ANY TIMINGS DONE.
02100
02200       THE  TIMINIT() ROUTINE  IS  CALLED  FROM  TIMSET()  AND
02300   PERFORMS  THE  FOLLOWING  ACTIONS:  1)  IT  TURNS  OFF   THE
02400   WRITE-PROTECT  BIT  IN  THE  HIGH SEGMENT; 2) IT INITIALIZES
02500   CERTAIN COUNTERS AND CREATES AN ENTRY IN THE TIME VECTOR FOR
02600   THE   MAIN   PROGRAM;   3) SCANS THE DDT SYMBOL TABLE SEARCHING
02700   FOR ROUTINE NAMES (A NAME WHICH SATISFIES CERTAIN  CRITERIA,
02800   BEST DISCOVERED BY EXAMINING THE CODE); 4) CREATING AN ENTRY
02900   IN THE TIME VECTOR FOR EACH ROUTINE FOUND; 5) REPLACING  THE
03000   "PUSH  SREG,FREG"  INSTRUCTION  AT  THE  BEGINNING  OF  EACH
03100   ROUTINE BY A "PUSHJ SREG,TIMENT" INSTRUCTION AND EVERY "POPJ
03200   SREG,"   INSTRUCTION    AT    THE    END  BY  A  "JRST  TIMEXT"
03300   INSTRUCTION: AND FINALLY 6)  IT  RESTORES  THE  HIGH-SEGMENT
03400   WRITE-PROTECT BIT TO ITS PREVIOUS STATUS.
03500
03600       THE TIMING UPON ROUTINE ENTRY IS CALCULATED AS FOLLOWS:
03700
03800           GRAB TIMER (DONE IN TIMENT)
03900           COMPUTE LOST TIME
04000           ADD TIME INCREMENT TO  ALL  ACTIVE  ROUTINES'  TOTAL
04100              TIME
04200           ADD TIME INCREMENT TO CURRENT ROUTINE TIME
04300           PUSH  THE NEWLY-ENTERED ROUTINE TIME VECTOR ONTO THE
04400              TIME STACK
04500           ADD 1 TO THE NUMBER OF CALLS
04600           GRAB TIMER (AGAIN DONE IN TIMENT)
04700
04800   THE TIMING UPON ROUTINE EXIT IS CALCULATED AS FOLLOWS:
04900
05000           GRAB TIMER (DONE IN TIMEXT)
05100           COMPUTE LOST TIME
05200           ADD  TIME  INCREMENT  TO  ALL ACTIVE ROUTINES' TOTAL
05300              TIME
05400           ADD TIME INCREMENT TO CURRENT ROUTINE'S ROUTINE TIME
05500           POP THE TIME VECTOR OF THE CURRENT ROUTINE
05600           GRAB TIMER (DONE IN TIMEXT)
05700
05800       LOST TIME IS THE TIME BETWEEN THE "GRAB TIMER" BEGINNING
05900   A TIMING ROUTINE AND THAT AT ITS END.
06000                                   8-10
```

```
06100       THE TIME VECTOR IS THE TABLE CONTAINING THE NAMES OF ALL
06200   ROUTINES IN THE SYSTEM, AND AREAS TO ACCUMULATE STATISTICS
06300   FOR THEM.  IT IS CURRENTLY 6 WORDS PER ENTRY TIMES THE
06400   NUMBER OF ENTRIES (MAXRTN) IN SIZE.
06500
```

- - - L O C A L I Z A T I O N M E A S U R E S - - -

LOCALIZATION MEASURES PROVIDE INFORMATION ABOUT THE DYNAMIC BEHAVIOR OF A PROGRAM WITH REGARDS TO ITS EXECUTION WITHIN CERTAIN REGIONS OF MEMORY AND ITS DATA ACCESSES. THE TIMER PACKAGE CANNOT OBTAIN STATISTICS ABOUT ITS BEHAVIOR WITH REGARD TO DATA ACCESSES, BUT IT CAN MONITOR THE INSTRUCTION LOCALIZATION. THESE MEASURES ARE USEFUL FOR DETERMING THE PROPER GROUPING OF ROUTINES OR MODULES FOR PAGING OR OVERLAYING.

THE LOCALIZATION STATISTICS OBTAINED ARE SOMEWHAT APPROXIMATE, SINCE THE ROUTINE IS AWARE ONLY OF THE BLOCK OF MEMORY WHICH CONTAINS THE ROUTINE ENTRY POINT. IF THE ROUTINE CROSSES A BLOCK BOUNDARY, THIS SHOULD COUNT AS A BLOCK CROSSING, BUT DOES NOT. IT WOULD BE HOPED THAT A BLISS VERSION FOR A PAGED PDP-10 WOULD HAVE A FACILITY TO FORCE ROUTINES TO THE NEXT PAGE BOUNDARY, RATHER THAN SPLIT THEM.

THE LOCALIZATION MONITOR RECORDS 1) THE NUMBER OF TIMES A BLOCK WAS ENTERED (A ROUTINE WITHIN THE BLOCK WAS CALLED) FROM A DIFFERENT BLOCK AND 2) THE NUMBER OF TIMES A ROUTINE WITHIN THE BLOCK CALLED A ROUTINE IN A DIFFERENT BLOCK, FROM THE REMAINDER OF THE TIMING INFORMATION, THE TOTAL NUMBER OF CALLS WHICH WERE MADE TO ROUTINES WITHIN THE MEMORY BLOCK AND THE TIME SPENT IN THESE ROUTINES IS OBTAINED. A SUPPORT ROUTINE PRINTS OUT A MEMORY MAP LISTING THE ROUTINES WITHIN EACH BLOCK. MORE SOPHISTICATED ANALYSIS IS POSSIBLE BY EITHER PROGRAM OR HUMAN.

```
00100   - - - - - L O C A L I Z A T I O N   O U T P U T - - - - -
00200
00300       THE LOCALIZATION OUTPUT CONSISTS OF TWO ROUTINES, TIMST8
00400   AND TIMST9, WHICH MAY BE CALLED FROM THE USER'S MAIN PROGRAM
00500   OR VIA DDT IN THE SAME MANNER THAT THE OTHER REPORTING
00600   ROUTINES ARE CALLED (SEE "USE FROM DDT" ABOVE).
00700
00800   TIMST8() OUTPUT:
00900
01000   BLOCK    IN       OUT      CALLS    TIME
01100   ******  ****    ******    ******* *******
01200
01300
01400       "BLOCK" IS THE MEMORY ADDRESS OF THE MEMORY BLOCK.  IF
01500   NO TRANSFERS IN OR OUT WERE MADE, THEN THIS IS THE ONLY
01600   INFORMATION ON THE LINE.
01700
01800       "IN" IS THE NUMBER OF CALLS MADE TO ROUTINES IN THE
01900   BLOCK FROM ROUTINES OUTSIDE THE BLOCK.
02000
02100       "OUT" IS THE NUMBER OF CALLS MADE TO ROUTINES OUTSIDE
02200   THE BLOCK FROM ROUTINES WITHIN THE BLOCK.
02300
02400       "CALLS" IS THE TOTAL NUMBER OF CALLS MADE TO ALL
02500   ROUTINES WITHIN THE BLOCK, FROM ALL OTHER ROUTINES
02600   ("CALLS"-"IN" GIVES SOME MEASURE OF THE INTRA-BLOCK
02700   ACTIVITY).
02800
02900       "TIME" IS THE TOTAL TIME SPENT EXECUTING ALL ROUTINES
03000   WITHIN THE BLOCK.  THIS IS THE SUM OF ALL "ROUTINE" TIMES
03100   FOR THE ROUTINES WITHIN THE BLOCK.
03200
03300
03400   TIMST9() OUTPUT:
03500
03600   BLOCK    RTNS
03700   ******  ******  ******  ******  ******  ******  ******
03800           ******  ******  ETC.
03900
04000       "BLOCK" IS THE MEMORY ADDRESS OF THE MEMORY BLOCK.
04100
04200       "RTNS" IS THE NAMES OF ALL ROUTINES CONTAINED IN THAT
04300   BLOCK.
04400
```

```
00100   - - - - I N T E R N A L   D O C U M E N T A T I O N - - - -
00200
00300   DATA GATHERING ROUTINES:
00400
00500   TIMENT: ALL ROUTINES LINK TO THIS ROUTINE UPON ROUTINE
00600           ENTRY. THE CLOCK IS READ AND ITS VALUE IS PASSED TO
00700           TIMEIN(). ON RETURN TO TIMENT THE CLOCK IS READ
00800           AGAIN TO PREVENT THE TIME SPENT IN THE TIMING
00900           ROUTINES FROM BEING COUNTED IN THE ROUTINE TIMES.
01000           THIS VALUE IS STORED IN THE GLOBAL VARIABLE
01100           "TIMPRE". TIMENT IS CONTAINED IN TIMINT.MAC AND
01200           USES 14 WORDS.
01300
01400   TIMEXT: ALL ROUTINES LINK TO THIS ROUTINE UPON ROUTINE EXIT.
01500           THE CLOCK IS READ AND ITS VALUE IS PASSED TO
01600           TIMEOUT(). ON RETURN TO TIMEXT THE CLOCK IS READ
01700           AGAIN TO PREVENT THE TIME SPENT IN THE TIMING
01800           ROUTINES FROM BEING COUNTED IN THE ROUTINE TIMES.
01900           THIS VALUE IS STORED IN THE GLOBAL VARIABLE
02000           "TIMPRE". TIMEXT IS CONTAINED IN TIMINT.MAC AND
02100           USES 11 WORDS.
02200
02300   TIMEIN: THIS ROUTINE IS CALLED FROM TIMENT AND IS PASSED THE
02400           ROUTINE ADDRESS (PLUS ONE) AND THE CURRENT TIME.
02500           THE INCREMENT OF TIME SINCE THE LAST READING OF THE
02600           CLOCK IS COMPUTED AND TIMACC() IS CALLED TO ADD IT
02700           TO ALL THE CUMULATIVE TIMES OF ALL OUTSTANDING
02800           ROUTINES. THE INCREMENT IS ALSO ADDED TO THE
02900           ROUTINE TIME OF THE CALLING ROUTINE. A TIME VECTOR
03000           POINTER TO THE TIME VECTOR OF THE CALLED ROUTINE IS
03100           PUSHED ONTO THE TIME STACK AND THE NUMBER OF CALLS
03200           IS INCREMENTED. ASSORTED STATISTICS ABOUT STACK
03300           DEPTH, NESTING DEPTH, AND BOUNDARY CROSSINGS ARE
03400           OBTAINED. USES 71 WORDS. CALLS TIMACC, TIMTRX,
03500           TIMERR, TIMLOC.
03600
03700   TIMEOUT:        THIS ROUTINE IS CALLED FROM TIMEXT AND IS
03800           PASSED THE CURRENT TIME. THE INCREMENT OF TIME
03900           SINCE THE LAST READING OF THE CLOCK IS COMPUTED AND
04000           TIMACC() IS CALLED TO ADD IT TO THE CUMULATIVE TIMES
04100           OF ALL OUTSTANDING ROUTINES. THE INCREMENT IS ALSO
04200           ADDED TO THE ROUTINE TIME OF THE CURRENT ROUTINE.
04300           THE TIME VECTOR POINTER OF THE CURRENT ROUTINE IS
04400           POPPED FROM THE TIME STACK. USES 23 WORDS. CALLS
04500           TIMACC.
04600
04700   TIMACC: THIS ROUTINE IS CALLED FROM TIMEIN AND TIMEOUT AND
04800           IS PASSED THE TIME INCREMENT TO BE ADDED. THIS
04900           INCREMENT IS ADDED TO THE CUMULATIVE TIME OF ALL
05000           ROUTINES POINTED TO BY POINTERS IN THE TIME STACK,
05100           TAKING CARE NOT TO ADD THE VALUE TWICE TO ROUTINES
05200           CALLED RECURSIVELY. USES 26 WORDS.
05300
05400   TIMLOC: THIS ROUTINE IS CALLED TO LOCATE THE TIME VECTOR OF
05500           THE ROUTINE BEING CALLED. IT RETURNS AS ITS VALUE
05600           THE INDEX OF THIS ROUTINE IN THE TIMEVECTOR
05700           STRUCTURE. USES BINARY SEARCH TECCNIQUE. USES 25
05800           WORDS.
05900
06000   TIMTRX: THIS ROUTINE IS CALLED FROM TIMEIN TO RECORD
```

```
06100           BOUNDARY  CROSSINGS.  TAKES  TWO  PARAMETERS,  THE
06200           ADDRESS OF THE CALLED ROUTINE AND THE ADDRESS OF THE
06300           CALLING ROUTINE.  IF THEY ARE IN DIFFERENT BLOCKS, A
06400           TRANSITION OUT OF THE CALLERS BLOCK AND ONE INTO THE
06500           CALLED BLOCK ARE RECORDED.  USES 17 WORDS.
06600
06700
06800   INITIALIZATION ROUTINES:
06900
07000   TIMSET: CALLS  TIMINIT().  ON RETURN IT READS THE CLOCK AND
07100           STORES THE VALUE IN THE  GLOBAL  VARIABLE  "TIMPRE".
07200           THIS  ROUTINE  IS CONTAINED IN TIMINT.MAC AND USES 8
07300           WORDS.
07400
07500   TIMINIT:        THIS  ROUTINE  IS  CALLED  BY  TIMSET()  AND
07600           INITIALIZES THE SYSTEM BEING TIMED.  IT OBTAINS
07700           WRITE PRIVILEGES IN THE HIGH SEGMENT, PREPARATORY TO
07800           PLACING  TRAPS  IN  THE ROUTINES.  IT THEN CREATES A
07900           DUMMY ENTRY FOR THE  MAIN  PROGRAM  (FROM  WHICH  IT
08000           ASSUMES  IT  WAS  CALLED)  SO THE MAIN PROGRAM LOOKS
08100           LIKE  A  CALLING  ROUTINE.   SEVERAL  COUNTERS   AND
08200           SWITCHES  ARE  INITIALIZED.  THE DDT SYMBOL TABLE IS
08300           SCANNED, AND EACH SYMBOL REFERRING TO A LOCATION  IN
08400           THE  ADDRESS  SPACE  IS EXAMINED.  IF THE SYMBOL AND
08500           THE WORD IT POINTS TO SATISFY CERTAIN CRITERIA,  THE
08600           SYMBOL  IS  CONSIDERED  A  ROUTINE  NAME.  TIMFIX IS
08700           CALLED TO PLACE ROUTINE  ENTRY/EXIT  TRAPS,  AND  AN
08800           ENTRY   IN  THE  TIMEVECTOR  STRUCTURE  IS  CREATED.
08900           FINALLY,  THE  OLD   VALUE   OF   THE   HIGH-SEGMENT
09000           WRITE-PROTECT  BIT  IS RESET.  USES 179 WORDS, CALLS
09100           TIMFIX, TIMMAK, TIMSRC, TIM50X.
09200
09300   TIMFIX: THIS ROUTINE IS CALLED FROM TIMINIT  TO  SET  TIMING
09400           TRAPS   IN   THE   ROUTINE.    IT  IS  PASSED  THREE
09500           PARAMETERS: THE ADDRESS OF THE ROUTINE, THE NAME  OF
09600           THE   ENTRY-TRAP   ROUTINE,  AND  THE  NAME  OF  THE
09700           EXIT-TRAP ROUTINE.  THE FIRST INSTRUCTION IN THE
09800           ROUTINE  IS  REPLACED  BY  A "PUSHJ <ENTRY ROUTINE>"
09900           INSTRUCTION; THE POPJ  TERMINATING  THE  ROUTINE  IS
10000           REPLACED  BY  A  "JRST  <EXIT ROUTINE>" INSTRUCTION.
10100           USES 29 WORDS.
10200
10300   TIMMAK: THIS ROUTINE IS CALLED TO CREATE A NEW ENTRY IN  THE
10400           TIMEVECTOR  STRUCTURE.   IT IS PASSED THE ADDRESS OF
10500           THE ROUTINE AND ITS SIXBIT  NAME.   IF  ADDING  THIS
10600           ROUTINE  WOULD  CAUSE THE TIMEVECTOR STRUCTURE TO BE
10700           EXCEEDED, AN ERROR FLAG  IS  SET  AND  NO  ENTRY  IS
10800           CREATED.  USES 33 WORDS.
10900
11000   TIMSRC: THIS ROUTINE IS CALLED BY TIMINIT TO OBTAIN THE NAME
11100           OF THE MAIN PROGRAM.  IT  SEARCHES  THE  DDT  SYMBOL
11200           TABLE  FOR EXACT EQUALITY OF ITS PARAMETER.  USES 25
11300           WORDS.
11400
11500   TIM50X: TAKES A RADIX50  SYMBOL  (E.G.  A  DDT  SYMBOL)  AND
11600           RETURNS   AS   ITS  VALUE  THE  SIXBIT  NAME,  RIGHT
11700           JUSTIFIED.  USES 25 WORDS.
11800
11900   TIM506: TAKES A RADIX50  CHARACTER  AND  CONVERTS  IT  TO  A
12000           SIXBIT CHARACTER.  USES 27 WORDS.
```

```
12100
12200
12300   TERMINATION ROUTINES:
12400
12500   TIMEND: READS THE CLOCK AND CALLS TIMEOUT() TO FINISH THE
12600           TIMING OF THE MAIN ROUTINE.  THEN CALLS TIMTOT() TO
12700           FINISH COMPUTATION OF CERTAIN VALUES.     USES 7
12800           WORDS.  THIS ROUTINE IS CONTAINED IN THE MODULE
12900           TIMINT.MAC.
13000
13100   TIMTOT: CALLED BY TIMEND TO COMPUTE AVERAGE TIMES FOR EACH
13200           ROUTINE.  ALSO COMPUTES TOTAL EXECUTION TIME AND
13300           TOTAL NUMBER OF CALLS.  USES 57 WORDS.
13400
13500
13600   REPORTING ROUTINES:
13700
13800   TIMLPT: SETS THE LPT SWITCH TO DIRECT OUTPUT TO THE LINE
13900           PRINTER.
14000
14100   TIMTTY: RESETS THE LPT SWITCH TO DIRECT OUTPUT TO THE TTY.
14200
14300   TIMWLPT:        CALLED BY TIMPUT TO WRITE A CHARACTER ON THE
14400           LPT.  TWO PARAMETERS ARE PASSED: A FUNCTION CODE AND
14500           A CHARACTER, THE FUNCTION CODES ARE: 0/ OPEN THE LPT
14600           AND WRITE THE CHARACTER GIVEN, 1/ WRITE THE
14700           CHARACTER GIVEN, 2/ CLOSE THE LPT (CHARACTER
14800           IGNORED), USES 60 WORDS.
14900
15000   TIMPUT: WRITES THE SINGLE CHARACTER PASSED TO IT ON THE LPT
15100           OR THE TTY, AS DIRECTED BY THE LPT SWITCH.  USES 14
15200           WORDS.
15300
15400   TIMSPUT:        WRITES THE STRING PASSED TO IT (AS
15500           5-CHARACTER GROUPS) ON THE LPT OR TTY, VIA TIMPUT.
15600           USES 37 WORDS.
15700
15800   TIMCRLF:        WRITES A CARRIAGE-RETURN/LINE-FEED PAIR ON
15900           THE OUTPUT DEVICE.  USES 10 WORDS.
16000
16100   TIMTAB: WRITES A TAB ON THE OUTPUT DEVICE.  USES 7 WORDS.
16200
16300   TIMPR6: WRITES THE LEFT-JUSTIFIED SIXBIT CHARACTER STRING
16400           GIVEN ON THE OUTPUT DEVICE.  USES 21 WORDS.  CALLS
16500           TIMPUT.
16600
16700   TIMDE2: THIS ROUTINE IS CALLED TO DO NUMERIC OUTPUT.  IT IS
16800           PASSED 3 PARAMETERS, THE NUMBER TO OUTPUT, THE WIDTH
16900           TO OUTPUT IT, AND THE BASE TO CONVERT IT BY (2<=
17000           BASE <= 10).  CALLS TIMDE2 AND TIMPUT.  USES 31
17100           WORDS.
17200
17300   TIMDEC: THIS ROUTINE IS CALLED TO DO DECIMAL OUTPUT.  IT IS
17400           PASSED TWO PARAMETERS, THE VALUE AND THE WIDTH.
17500           CALLS TIMDE2 AND TIMPUT.  USES 40 WORDS.
17600
17700   TIMOCT: THIS ROUTINE IS CALLED TO DO OCTAL OUTPUT.  IT IS
17800           PASSED TWO PARAMETERS, THE VALUE AND THE WIDTH.
17900           CALLS TIMDE2 AND TIMPUT.  USES 40 WORDS.
18000
```

```
18100   TIMRE2: PUTS OUT THE STATISTICAL INFORMATION ABOUT TOTAL
18200           PERFORMANCE AND INDIVIDUAL ROUTINE PERFORMANCE.
18300           USES TIMPUT, TIMSPUT, TIMDEC, TIMPR6, TIMCRLF,
18400           TIMTAB, TIMWLPT. USES 284 WORDS.
18500
18600   TIMST1: SORTS DATA BY NAME AND CALLS TIMRE2, USES TIMESORT
18700           AND TIMRE2. USES 12 WORDS.
18800
18900   TIMST2: SORTS DATA BY CALLS AND CALLS TIMRE2, USES TIMESORT
19000           AND TIMRE2. USES 12 WORDS.
19100
19200   TIMST3: SORTS DATA BY ROUTINE TIME AND CALLS TIMRE2. USES
19300           TIMESORT AND TIMRE2. USES 12 WORDS.
19400
19500   TIMST4: SORTS DATA BY CUMULATIVE TIME AND CALLS TIMRE2.
19600           USES TIMESORT AND TIMRE2. USES 12 WORDS.
19700
19800   TIMST5: SORTS DATA BY AVERAGE ROUTINE TIME AND CALLS TIMRE2.
19900           USES TIMESORT AND TIMRE2. USES 12 WORDS.
20000
20100   TIMST6: SORTS DATA BY AVERAGE CUMULATIVE TIME AND CALLS
20200           TIMRE2. USES TIMESORT AND TIMRE2. USES 12 WORDS.
20300
20400   TIMST7: SORTS DATA BY ADDRESS AND CALLS TIMRE2. USES
20500           TIMESORT AND TIMRE2. USES 12 WORDS.
20600
20700   TIMST8: CALLS TIMSCN, SPECIFYING TIMTRP AS THE PROCESSING
20800           ROUTINE. USES 8 WORDS.
20900
21000   TIMST9: CALLS TIMSCN, SPECIFYING TIMPRT AS THE PROCESSING
21100           ROUTINE. USES 8 WORDS.
21200
21300   TIMALL: CALLS TIMST1 THRU TIMST9. USES 14 WORDS.
21400
21500   TIMSCN: CALLS TIMESORT TO SORT DATA BY ADDRESSES. SEQUENCES
21600           THRU THE ADDRESS SPACE IN BLOCKS OF
21700           2**COREBLOCK, CALLING THE REQUESTED ROUTINES
21800           (PASSED BY ITS CALLER). USES TIMSPUT, TIMCRLF,
21900           TIMWLPT. USES 68 WORDS.
22000
22100   TIMTRP: FOR EACH CORE BLOCK FOR WHICH THERE IS A TRANSITION
22200           IN OR OUT, PRINT THE NUMBER OF EACH KIND, THE TOTAL
22300           TIME SPENT IN THE BLOCK, AND THE TOTAL NUMBER OF
22400           CALLS TO ROUTINES IN THE BLOCK. CALLS TIMSPUT,
22500           TIMDEC, TIMOCT, TIMCRLF. USES 79 WORDS.
22600
22700   TIMPRT: FOR EACH BLOCK, PRINTS OUT THE NAMES OF THE ROUTINES
22800           IN THAT BLOCK. USES TIMOCT, TIMTAB, TIMPR6,
22900           TIMCRLF. USES 50 WORDS.
23000
23100   MISCELLANEOUS ROUTINES:
23200
23300   TIMESORT:       SORTS THE TIMEVECTOR DATA BY CREATING A
23400           SORTED INDEX VECTOR INTO THE TIMEVECTOR. THE SORT
23500           FIELD IS SPECIFIED BY THE PARAMETERS. THE 5
23600           PARAMETERS REQUIRED ARE: 1) NUMBER OF ENTRIES TO BE
23700           SORTED; 2) WHICH WORD OF THE TIMEVECTOR TO SORT ON;
23800           3,4) THE POSITION (3) AND SIZE (4) FIELD
23900           SPECIFICATIONS OF THE BYTE OF THE WORD TO SORT ON;
24000           5) THE DIRECTION TO SORT. THE ALGORITHM IS A
```

```
24100           GENERALIZATION OF FLOYD'S TREESORT 3 (ALGORITHM 245,
24200           CACM  DEC, 1964).  IT CONTAINS FOR ITS EXCLUSIVE USE
24300           THE ROUTINES TIMSIFT(58 WORDS) , TIMCMP (30  WORDS),
24400           TIMXFR  (11  WORDS),  TIMEXCH  (19  WORDS).  TIMESORT
24500           ITSELF IS 53 WORDS LONG.
24600
24700   TIMSIFT:        SEE TIMESORT.
24800
24900   TIMCMP: SEE TIMESORT.
25000
25100   TIMXFR: SEE TIMESORT.
25200
25300   TIMEXCH:        SEE TIMESORT.
25400
25500   TIMERR: A GENERAL ERROR-CATCHER,  ANY ERROR  DETECTED  CALLS
25600           TIMERR,  PASSING IT AN ERROR CODE,  THE CODE IS USED
25700           IN A SELECT EXPRESSION  TO  CHOOSE  THE  APPROPRIATE
25800           ACTION.  USES 48 WORDS.
25900
```