2.9. JOVIAL/J3B

2.9.1. LANGUAGE FEATURES

JOVIAL/J3B [REI75,SOF75] is a high level language developed by SofTech for use in avionics applications. The language is based on JOVIAL/J3, the Air Force standard language for command and control applications. JOVIAL/J3B has been used extensively in the B-1 Strategic Bomber program.

A. Basic Data Types and Operators

JOVIAL/J3B has seven basic data types: signed integer, unsigned integer, fixed point, single and double precision floating point, bit string, and character string. The length of bit strings is limited to the implementation dependent number of bits in a computer word. Automatic conversion is performed between integer and floating point expressions. The following types of literals are permitted in JOVIAL/J3B expressions: integer, fixed point, single and double precision floating point, and hexadecimal and character strings.

The operators and the data types on which they operate are listed below:

arithmetic operators

+, -, *, /, **

ABS - Absolute value.

INTR - Extracts a bit string from an arbitrary expression and converts the string to integer.

INTGR - Converts a fixed point expression to integer.

FIX - Converts an integer expression to fixed point.

SCALE - Scales a fixed point expression.

relational operators

=, <>, <, >, <=, >=

- All the relational operators can be used to compare numeric expressions, but character expressions can only be compared using = and <> (there is no

explicit ordering of the character set). Bit expressions can not be compared using the relational operators.

All the relational operators yield a bit string result.

bit operators

NOT, AND, OR, XOR

SHIFTL - Left shift.

SHIFTR - Right shift.

BIT - Pseudo-variable for accessing bit strings. The BIT function can appear on the left-hand side of an assignment statement.

character operators

BYTE - Pseudo-variable for accessing a sequence of bytes (characters). Can appear on left-hand side of an assignment statement.

B. Control Structures

- BEGIN <stmt-list> END ;
  (Compound statement.)

- IF <bit-expr> ; <stmt> ; { ELSE <stmt> } ;
  (Standard conditional statement with optional ELSE part. If the value of the <bit-expr> is known at compile time then no code is generated for the bypassed THEN or ELSE <stmt>.)

- WHILE <bit-expr> ; <stmt> ;
  (Standard while loop.)

- FOR <var> (<init-expr> BY <incr-expr>
  { WHILE <bit-expr> }); <stmt> ;
  (For loop with optional WHILE clause.)

- FOR <var> (<init-expr> THEN <next-expr>
  { WHILE <bit-expr> }); <stmt> ;
  (Alternate form of for loop. The <var> is assigned

the value of the <init-expr> on the first iteration of the loop, and the value of <next-expr> on all subsequent iterations. The <next-expr> can be any integer expression, and it is evaluated on each iteration of the loop.)

- GOTO <label> ;

  (Unconditional branch to label in current namescope.)

- GOTO <switch-name> (<integer-expr>) ;

  (Computed goto. The <switch-name> must have been declared with a statement of the form

  SWITCH <switch-name> = <label-list> ;

  On execution of the GOTO statement the value i of the <integer-expr> is used to select the i-th label, and a branch is made to the selected label.)

- RETURN ;

  (Return from a procedure or function.)

- { DEF } { RENT } PROC <proc-name>
  ({ <input-parameters> } { : <output-parameters> })
  { <function-type> } ;
  BEGIN
  <local-declarations> ;
  <stmt-list> ;
  END ;

  (Definition of a procedure or function. If the RENT option is selected then reentrant code will be generated for the procedure (recursive calls are not permitted, however). The DEF option permits the <proc-name> to be called from other external procedures. Any labels appearing in the <stmt-list> must be declared in the <local-declarations> section.)

- <proc-name> ({ <input-arguments> }
  { : <output-arguments> }) ;

  (Invocation of a procedure or function. The method used to pass parameters (such as call by value,

value-result, or reference) is implementation dependent.

Internal procedures can be declared as "inline" routines with a statement of the form INLINE <proc-name> ; Each invocation of an inline procedure causes the body of procedure to be substituted inline at the point of invocation.)

C. Data Structures

JOVIAL/J3B has three constructs for creating more complex data structures from the basic data types:

(a) arrays

Arrays are declared with a statement of the form

ARRAY <var-name> (<dimension-list>) <type> ;

The array type can be any of the basic data types or a pointer to a table. Arrays can have up to three dimensions, and array indexing starts at 0. The elements in an array are referenced using the standard subscript operator <var-name> (<subscript-list>) .

(b) tables

The JOVIAL language has extensive facilities for constructing data tables (linear lists of record structures). A "template" for the entries in a table is declared using the TYPE statement:

TYPE <new-template> TABLE () { $\overset{M}{D}$ }
{ LIKE <old-template> }
BEGIN <item-declarations> END ;

The options M and D affect the packing density of the items within the individual table entries (M-medium, D-dense). The LIKE option allows a previously defined table template to be used in the definition of a new template. The items in the <new-template> will consist of all items in the <item-declarations> list, preceded by the items in the <old-template> if the LIKE option was used. The type of the

items in the <item-declarations> list can be any of the basic data types or a pointer to a table.

The table templates can then be used to declare data tables:

TABLE <table-name> (<number-of-entries>) { P } { M D } ;
BEGIN <item-declarations> END ;

The M and D options have already been described. The default method for allocating storage for a table is by table entry: for each table entry there is a contiguous block of core that is long enough to contain all the items in the <item-declarations> list. If the P option is specified, however, the table is allocated in a "parallel" fashion: there is a contiguous block of core for the first item in all the table entries, a block for all the second items, and so forth.

An alternate version of the table declaration gives the programmer complete control over placement of items within a table entry. The number of words per table entry and the placement of each item (word position and starting bit within the word) is directly specified. The storage for items can overlap.

Individual items in a directly declared table (declared without a template) are accessed using subscript notation:

<table-item> (<table-entry>)

An entire table entry can be compared with or assigned the value of another table entry using the ENTRY function. For example,

ENTRY(MSG.TABLE(I)) = ENTRY(ERROR.MSG(4)) ;

Table entries or items within a table entry of any table declared with a template can only be accessed using pointers. Pointers will be discussed in the next section.

Note: There are a large number of restrictions on the way that tables can be declared and used.

(c) pointers to table entries

A pointer is declared with a statement of the form

ITEM <pointer-var> P { <template> } ;

Pointers declared with a template can only be used to access table entries having the same template, pointers declared without a template can point to any entry.

The following pointer functions and operators are available:

POINT(<table-name>, <subscript>)

- Yields a pointer to the specified entry in the table.

NEXT(<entry-pointer>, <table-name>, { <index> })

- Yields a pointer to the next table entry following <entry-pointer>. The <index> can be used to obtain a pointer to some table entry relative to <entry-pointer>. For example,

  NEXT(MSGPTR,MSG.TABLE)

  NEXT(MSGPTR,MSG.TABLE,-2) .

<table-item> (<entry-pointer>)

- Accesses the specified item in the table entry pointed at by the <entry-pointer>.

The relational operators =, <>, <, >, <=, >= can be used to compare compatible pointers, and the assignement operator = can be used to copy a pointer.

D. Other Features

JOVIAL/J3B has a number of features that would be helpful for programming large systems. Source files containing JOVIAL statements can be inserted into a program using a statement of the form COPY <file-spec> ; . Program constants can be declared using the CONSTANT statement:

CONSTANT <constant-name> <type> = <value> ;

The <constant-name> can be used in any expression, but it can not be assigned a new value by an assignment statement or a procedure call.

The language also has a simple replacement and a parameterized macro facility. Macros are declared with the

statement

DEFINE <macro-name> ((<parameter-list>))
<replacement-string> ;

The <replacement-string> can contain other macros.

JOVIAL/J3B has a COMPOOL feature that is similar to (but more awkward than) the HAL/S COMPOOL block. A JOVIAL COMPOOL file can contain constant and macro definitions, declarations of external procedures and functions, templates for tables, and references to BLOCK definitions (BLOCK definitions are used for declaring shared, external data; they are a combination of the Fortran COMMON and BLOCK DATA statements.) The COMPOOL file can be invoked by any program that requires the declarations and templates.

The language has an OVERLAY statement that is similar to the Fortran EQUIVALENCE statement. JOVIAL/J3B has no I/O facilities.

E. Runtime Environment

JOVIAL/J3B requires a runtime stack for any procedures declared to be reentrant. Other than this, the language requires little in the way of runtime environment.

F. Syntax

The BNF grammar for JOVIAL/J3B has approximately 500 productions (the SofTech grammar for the language includes type restrictions and is considerably more precise than typical BNF grammars).

2.9.2. CHARACTERISTICS

A. Machine Dependence

JOVIAL/J3B has a large number of implementation dependent features, including the method used to pass procedure and function parameters, the maximum length of bit strings (limited to one computer word), the functions INTR, BIT, and BYTE used for accessing bit and character strings, the OVERLAY statement,

programmer specified table allocation (word position and bit position within a word), the restriction that all items in a table declared with the parallel (P) attribute occupy a single word, and the lack of a collating sequence for the character set.

B. Efficiency

The language should be as efficient as Fortran for programs using non-reentrant procedures. All data areas can be allocated statically.

C. Level of the Language

JOVIAL/J3B is a high level language.

D. Size of the Language and Compiler

JOVIAL/J3B is a large language with complicated data structures (the TABLE in particular). The compiler will also be large.

E. Special System Features

The language has bit and character data types, the INTR, BIT, and BYTE functions for accessing bits and characters, reentrant procedures and functions, the OVERLAY statement for equivalencing data storage, and the TABLE data structure. All of these features would be helpful for system programming.

F. Error Checking and Debugging

The JOVIAL/J3B language is strongly typed, so many program errors can be detected during compilation.

As discussed in section B. Control Structures, no code is generated for the bypassed section of an IF statement when the value of the <bit-expr> is known at compile time. This feature permits debugging code to be maintained in a JOVIAL/J3B program without any expense in execution time or space. For example:

```
DEFINE DEBUG = X'1';
    .
    .
    .
IF DEBUG ; BEGIN <debug-statements> END;
```

The language manual does not indicate that any other debugging features are available.

G. Design Support

(a) modularity

Modularity in JOVIAL/J3B is good. The language has procedures, functions, and the basic control structures for structured programming. The language permits independent compilation of procedures, functions, and COMPOOL files. The COMPOOL and COPY files can be used to store commonly used declarations or source text, and the BLOCK statement permits sharing of external data .

(b) modifiability

JOVIAL/J3B has a number of features which would aid in program modification, including the CONSTANT declaration, the DEFINE statement for defining macros, the COPY statement for including source files, and the COMPOOL files. The language also has a structured control structure which will tend to make programs more readable.

(c) reliability

The pseudo-variables BIT and BYTE for accessing bit and character strings need to be used carefully, since they can be used to alter any portion of a data item. The OVERLAY statement and user-specified table allocation also require careful programming. In general, however, it should be considerably easier to write reliable programs in JOVIAL/J3B than in a language like fortran. JOVIAL/J3B has structured control structures, array and table data structures, a large set of basic data types, several features that can improve the readability of programs (macros and CONSTANT items), COMPOOL and COPY files to insure that separately compiled programs employ the same data declarations, and strong type checking.

H. Use

JOVIAL/J3B has been implemented on the IBM 370 series and a number of special purpose minicomputers including the SKC 2070, SKC 2000, IBM 4T , and the LITTON 4516D. The compiler was developed by SofTech using the AED language. JOVIAL/J3B has been used extensively in the B-1 Strategic Bomber program.

2.10. LITTLE

2.10.1. LANGUAGE FEATURES

LITTLE [SHI74] was developed at NYU in 1968 in an attempt to produce an efficient but machine independent systems implementation language. The only data type supported by the language is bit strings of arbitrary (but not varying) length, and no type checking is performed. LITTLE is essentially a Fortran language with some structured programming constructs.

A. Basic Data Types and Operators

LITTLE is a typeless language that operates on bit strings of arbitrary length. The language allows five types of constants to appear in expressions: unsigned integers, octal numbers, binary numbers, mixed binary/octal numbers, and character strings (including the empty string). Note that floating point numbers are not provided. The following operators are provided:

bit string operators

.OR., .AND., .EXOR.

Bitwise OR, AND, and exclusive OR of two expressions. The shorter operand is padded on left with zeros.

.NOT.

.FB.

Position of leftmost 1-bit in expression.

.NB.

Number of 1-bits in expression.

.C.

Bitwise concatenation of two operands.

.E. <start bit> <number of bits> <expr>
.F.

Pseudo variables for inserting or extracting bits. The .E. operator must be used for operands extending across word boundarys.

arithmetic operators

+, -, *, /

Integer arithmetic operators.

relational operators

.EQ., .NE., .LT., .GT., .LE., .GE.

character operators

<string-1> .IN. <string-2>

Index of <string-1> in <string-2>.

.S. <start character> <number of characters> <string>

Pseudo variable for inserting or deleting character strings.

.CH. <character number> <string>

Pseudo variable for inserting or deleting single characters.

<string-1> .CC. <string-2>

String concatenation.

B. Control Structures

- IF (<expr>) <stmt> ;

(Simple if statement.)

- IF <expr> THEN <stmt-list> { ELSE <stmt-list> } END IF;

(Compound if.)

- WHILE <expr> ;
  UNTIL

<stmt-list>

END WHILE ;
UNTIL

(Standard while and repeat loops.)

- DO <var> = <expr-1> TO <expr-2> BY <expr-3> ;

<stmt-list>

END DO ;

(Standard for loop.)

- GOTO <label> ;

GOBY (<expr>) (<label-1>, ..., <label-k>) ;

(Unconditional and computed goto.)

- SUBR <ident> { (<parameter-list>) } ;

<stmt-list>
END SUBR ;

FNCT <ident> [(<parameter-list>)] ;
<stmt-list>
END FNCT ;

(Fortran like subroutines and functions. Neither can be recursive. A function may not assign values to its input parameters.)

- CONT [<specifier>] ;

(Continue next iteration of the innermost or specified DO, WHILE, or UNTIL loop.)

- QUIT [<specifier>] ;

(Exit the innermost or specified loop.)

- RETURN ;

(Return from a subroutine or function.)

C. Data Structures

The only data structure supported by LITTLE is the one-dimensional array. The statements

SIZE <ident> (<length in bits>) ;
DIMS <ident> (<number of elements>) ;

declares <ident> to be a one-dimensional array, each element of which is a bit string of length <length in bits>. Array elements are accessed using standard subscript notation:

<ident> (<subscript>) .

D. Other Features

LITTLE is a typeless, Fortran-like language with no block structure and comments in /* */ or $ to end-of-line pairs. LITTLE has a DATA statement for initializing variables, and a NAMESET feature similar to Fortran COMMON. The language also has a simple and a parameterized macro facility allowing recursive macro expansion.

E. Runtime Environment

Because LITTLE forbids recursive subroutines or functions, the language does not require a runtime stack. There is also no need for any form of dynamic storage allocator.

F. Syntax

The BNF grammar for LITTLE has approximately 80 productions.

2.10.2. CHARACTERISTICS

A. Machine Dependence

LITTLE has no machine dependent features and could be implemented on most machines. However, because of the arbitrary length of operands, there are few machines that could implement LITTLE efficiently for operands longer than the word size.

B. Efficiency

LITTLE should be efficient for programs using variables that match the word size of the host machine. Inline code can be generated for most operators, there is no need for a runtime stack, and there is no block entry or dynamic storage allocation. For expressions involving operands longer than a single word, however, LITTLE may execute considerably slower than hand-coded assembly language.

C. Level

LITTLE is a low-level language.

D. Size of Language and Compiler

LITTLE is a small language, and the compiler should also be small.

E. Special System Features

None.

F. Error Checking and Debugging

Because of the lack of data types, LITTLE can perform no compile or runtime type checking. Other runtime checks, such as subscript errors or expression out of range in a GOBY statement, will be performed if the debug option is specified.

The CDC 6600 implementaion of LITTLE provides the following debugging facilities: (1) trace of assignments to selected variables; (2) calling history of subprograms; (3) statistics on number of statements executed by statement type; (4) subscript checks for arrays; and (5) verification that certain assertions (LITTLE expressions involving program variables) are true.

G.Design Support

(a) modularity

LITTLE allows independent compilation of modules, and provides communication through NAMESET (Fortran COMMON) blocks.

(b) modifiability

LITTLE has a fairly powerful macro processor, the standard structured programming constructs, and a feature for conditional compilation of source text. These would be a great help in modifying LITTLE programs. However, the lack of any features for constructing new data types (other than one-dimensional arrays) means that all data structures would have to be implemented by the LITTLE programs themselves. Subsequent changes to the data structures could require large scale revisions of the program.

(c) reliability

Because LITTLE is a typeless language, the compiler performs no compile or runtime checks to insure that the bit pattern in an operand is meaningful. Type checking is therefore the user's responsibility. In addition, the lack of data structures requires LITTLE programs to simulate the data structures with LITTLE statements. LITTLE programs will then be longer, more complex, and harder to understand than a program written in a language with more data structuring facilities.

H. Use

LITTLE has been implemented on the CDC 6600, the IBM 360 series, and the Honeywell 512. The compiler is written in LITTLE itself, and could easily be bootstrapped onto other machines.

2.11. PASCAL

2.11.1. LANGUAGE FEATURES

PASCAL [JEN74,RIC76] is a general purpose, high level language designed by Niklaus Wirth as a successor to ALGOL 60. The language has a full set of control structures for structured programming, and many facilities for data structuring including arrays, records, sets, and typed pointers. PASCAL has been used for a number of systems-oriented problems including the compilers for PASCAL and CONCURRENT PASCAL, and the SOLO operating system (a single-user operating system for the PDP 11/45).

A. Basic Data Types and Operators

PASCAL has four basic data types: INTEGER, REAL, BOOLEAN, and CHAR (single character). Full type checking is performed at compile time, and no automatic conversions are performed between the basic types. The following types of constants are permitted in expressions: integer, real, boolean, character, and string (treated as an array of characters).

The operators and the data types on which they operate are listed below:

arithmetic operators and functions (INTEGER and REAL operands)

+, -, *, /	- Standard arithmetic operators for INTEGER or REAL operands. The division operator returns a REAL result.
DIV, MOD	- Division and modulus operators for INTEGER operands.
ABS(<expr>)	- Absolute value of REAL or INTEGER expression.
SQR(<expr>)	- Square of REAL or INTEGER <expr>.

The following functions are available for INTEGER operands:

ODD(<expr>)	- Function returning true if the expression

is odd.

SUCC(<expr>) - Functions yielding successor and
PRED(<expr>) predecessor of the expression.

The following functions are available for REAL operands:

TRUNC(<expr>) - Functions yielding INTEGER result of
ROUND(<expr>) truncating or rounding a REAL <expr>.
SIN, COS, - Standard mathematical functions.
ARCTAN, LN,
EXP, SQRT

logical operators (BOOLEAN operands)

AND, OR, NOT - The BOOLEAN operators yield a BOOLEAN result.

relational operators (all basic types)

=, <>, <, >, <=, >=
- The two operands must have the same type. The relational operators yield a BOOLEAN result.

character operators

SUCC, PRED - Successor and predecessor functions.
CHR(<expr>) - Yields i-th character in the character set, where i is the value of <expr>.
ORD(<char>) - Ordinal position of the character in the character set.

B. Control Structures

- BEGIN <stmt-list> END
  (Compound statement.)

- IF <boolean-expr> THEN <stmt> { ELSE <stmt> }
  (Standard conditional with optional ELSE clause.)

- WHILE <boolean-expr> DO <stmt>
  (While loop.)

- REPEAT <stmt-list> UNTIL <boolean-expr>

(Until loop. The body of the loop will be executed at least once.)

- FOR <var> := <expr-1> TO DOWNTO <expr-2> DO <stmt>

(For loops with implied increments of +1 and -1.)

- CASE <scalar-expr> OF
  <constant-list-1> : <stmt-1>
  . . .
  <constant-list-k> : <stmt-k>
  END

(Case statement. The <scalar-expr> can be INTEGER, CHAR, BOOLEAN, or any user-defined scalar or subrange type (scalar and subrange types will be described later in Section C). The constant lists must contain constants of the same type as the <scalar-expr>. The <scalar-expr> is evaluated, and the constant lists are scanned to find a constant equal to the expression. If a match is found then the corresponding statement is executed; if no match is found then none of the statements are executed.)

- WITH <variable-list> DO <stmt>

(Executes <stmt> using the record variables in the <variable-list.> Any expression in <stmt> may refer to subcomponents of the records without fully qualifying the subcomponent. For example, if X is a record with subcomponents A, B, and C, then

```
WITH X DO BEGIN
   A := A + 1.0;
   B := A < 10.0;
   C := 'G'
END
```

is equivalent to

```
X.A := X.A + 1.0;
X.B := X.A < 10.0;
```

```
        X.C := 'G';
    )
```

- GOTO <label>;

  (Unconditional transfer to a statement in the current namescope. PASCAL requires that all labels be declared with the LABEL statement.)

- PROCEDURE <proc-name> { (<parameter-list>) }; <proc-body>

  FUNCTION <func-name> { (<parameter-list>) } : <type>
  <func-body>

  (Procedure and function definitions. Both may be recursive. The user can request that parameters be passed by value or by reference.)

- <func-name> { (<argument-list>) }
  <proc-name> { (<argument-list>) }

  (Invoke a function or procedure.)

C. Data Structures

PASCAL has seven constructs for creating more complex data structures from the basic data types:

(1) scalar type

The scalar type statement

```
TYPE <type-ident> = (<object-1>, ..., <object-k>) ;
```

defines an ordered set consisting of <object-1>, ..., <object-k>. For example:

```
TYPE MONTH = (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,
              SEP,OCT,NOV,DEC) ;
```

The set is ordered, so the relational operators =, <>, <, >, <=, >=, the assignment operator :=, and the functions SUCC, PRED, and ORD can be applied to any scalar type. Note: the basic types INTEGER, CHAR, and BOOLEAN are predefined scalar types.

(2) subrange types

Subrange types are subranges of scalar types, and they

also form ordered sets of objects. The statement

```
TYPE <type-ident> = <object-1> .. <object-m> ;
```

defines a subrange type. There must be a scalar type containing both objects, and the first object must be less than the second. For example:

```
TYPE SPRING = MAR .. MAY;
TYPE DIGIT = '0' .. '9';
TYPE INDEX = 0 .. 100;
```

All the operators for scalar types can be applied to subrange types.

(3) arrays

The statement

```
TYPE <type-id> = ARRAY [<dimension-list>] OF <type> ;
```

defines an array type. Arrays can have an arbitrary number of dimensions, and the <type> can be any basic type or one of the types discussed in this section. The dimensions are specified by subrange types. For example:

```
TYPE MATRIX = ARRAY[1..3, 1..3] OF REAL;
VAR VECTOR : ARRAY[1..10] OF REAL;
VAR JOBSRUN : ARRAY[1968..1973, JAN..DEC] OF INTEGER;
```

The assignment operator := may be used to copy entire arrays, and array elements are referenced by listing the subscripts in brackets:
<ident> [<subscript-list>] .

(4) sets

The statement

```
TYPE <type-ident> = SET OF <base-type> ;
```

defines a type consisting of all possible subsets of the <base-type>, which must be a scalar or subrange type. For example:

```
TYPE DAY = (M,T,W,TH,F,SA,S);   {Define scalar type}
VAR DAYSOFF : SET OF DAY;       {Now use it for a set}
VAR DIGITS : SET OF 0..9;
```

The following operators are available for manipulating set types:

[<element-list>]	- Set constructor yielding set. The list may be empty.
+, -, *	- Set union, difference, and intersection.
=, <>	- Tests on equality or inequality.
<=, >=	- Tests on set inclusion.
IN	- Membership operator yielding true if element is in set.

(5) typed pointers

Pointer types are defined with a statement of the form

TYPE <type-ident> = ^ <type> ;

where <type> is any type. There is no "address" function in PASCAL - it is not possible to obtain the address of a variable. Instead, all pointers in PASCAL point into a dynamic storage area, and new pointers can only be created by requesting the allocation of some new data object in this storage area.

The following pointer operators and functions are available (Assume that pointer P is declared as VAR P : ^ X; .)

NEW(P)	- Allocates enough space for an object of type X, and sets P to the address of the space.
DISPOSE(P)	- Deallocates the object pointed to by P and sets P to NIL.
P ^	- Dereference operator yielding object pointed at by P. May appear on the left-hand side of an assignment statement.
=, <>	- Tests on pointer equality.
:=	- The assignment operator can be to copy pointers.

(6) file type

The statement

TYPE <type-ident> = FILE OF <type> ;

defines a sequential file of objects of type <type>. The declaration of a variable using this type (i.e., the declaration of a file) causes the implicit declaration of a variable X^, where X is the name of the file variable. This variable X^ has type <type>, and acts as the buffer pointer for the file. The basic file functions are

```
RESET(X)      - Sets X^ to the first record in the file X.
REWRITE(X)    - Prepares file X for rewriting.
GET(X)        - Gets the next record and assigns it to X^.
PUT(X)        - Writes out X^ into the file.
```

(7) record structures

A record type is declared with a statement of the form

```
TYPE <type-ident> = RECORD
  <member-1> : <type-1>
       .          .
       .          .
       .          .
  <member-k> : <type-k>
  { CASE <tag-field> : <type> OF
     <case-label-list-1> : (<variant-list-1>);
            .                      .
            .                      .
            .                      .
     <case-label-list-k> : (<variant-list-k>)  }
END ;
```

Records can contain an arbitrary number of members, and each member can be of any type. PASCAL records can also contain a "variant" part at the end of the record. This variant part permits records of the same type to contain a different number and different types of members. The value of the <tag-field> determines what is stored in the variant portion of the record. For example:

```
TYPE LINK = ^ PROCDESCRIPTOR;
     PRIORITY = 1..6;
     PROCDESCRIPTOR =
      RECORD      {Define a process descriptor.}
         FLINK,BLINK : LINK; {Forward/backward ptrs}
         GPR : ARRAY [0..7]; {General registers}
```

```
        PSW : INTEGER;          {Program status}
        CASE PR : PRIORITY OF {Variant part}
          1,2,3 : (MAXTIME,
                   MAXPAGES : INTEGER);
          4 : ( );
          5,6 : (BUFFER : ARRAY[0..128] OF INTEGER)
      END
```

The dot operator "." is used to reference members of a record. For example:

```
VAR P : PROCDESCRIPTOR;
VAR I : INTEGER;
FOR I := 0 TO 7 DO P.GPR[I] := 0;
P.PR := 2;
P.MAXTIME := 5;
P.MAXPAGES := 1000;
```

The WITH statement discussed in Section B can be used to avoid qualifying each member of a record with the record name. The assignment operator := can be used to copy an entire record.

D. Other Features

PASCAL requires the declaration of all variables, functions, procedures, and labels. PASCAL has a declaration of the form

```
CONST <ident> = <expr>;
```

for declaring program constants. The identifier can be used in any expression, but the value of the identifier can not be altered. PASCAL does not provide dynamic arrays or even array dimensions as parameters, as in the following FORTRAN segment:

```
      SUBROUTINE XYZ(ARRAY,N,M)
      INTEGER N,M,ARRAY(N,M)
```

Thus, it is not possible to write a PASCAL program that manipulates arrays of arbitrary sizes.

Finally, the language does not permit external functions or procedures: a PASCAL program consists of a main program and an arbitrary number of nested functions and procedures, and the entire program must be compiled as a unit.

E. Runtime Environment

PASCAL requires a runtime stack (all functions and procedures are potentially recursive), I/O routines, and a dynamic storage allocator. Some implementations may provide a garbage collector for repacking the dynamic storage area.

F. Syntax

PASCAL has a BNF grammar with approximately 150 productions.

2.11.2. CHARACTERISTICS

A. Machine Dependence

PASCAL is not machine dependent and has been implemented on a large number of machines.

B. Efficiency

PASCAL is moderately efficient. The language does require a runtime stack, and dynamic storage allocation is required in any program using pointers or files. However, the language features have been carefully selected to permit efficient implementation of the language. Sets can be represented by bits strings; the set union, intersection, and difference operators can then be implemented in just a few instructions. Scalar and subrange types are equivalently simple. The structured control structures also permit better code optimization.

C. Level

PASCAL is a high level language.

D. Size of the Language and Compiler

The PASCAL language is moderate in size. The compiler, which is written in PASCAL itself, is only 8500 statements.

E. Special System Features

PASCAL has typed pointers, dynamic storage allocation,

records, and the set type (which can be viewed as bit strings).

F. Error Checking and Debugging

PASCAL performs full type checking at compile time. In addition, pointers are fully typed and all pointers point into the dynamic storage area. This prevents pointers from pointing to objects of the wrong type, pointers containing illegal machine addresses, attempts to deallocate storage that was never allocated, or attempts to access data in deallocated areas. The subrange types also allow the implementation to perform runtime checks on variables to insure that the values are within the subrange. Such a feature would be very helpful in a diagnostic compiler.

The PASCAL manual does not indicate that any special debugging tools are available.

G. Design Support

(a) modularity

Modularity in PASCAL is fair. The language has a full set of structured control structures, and internal procedures and functions are provided. However, PASCAL does not permit external procedures or functions. This makes it costly to use existing programs (in a system library, for example), since the programs must be recompiled each time they are used.

(b) modifiability

As discussed previously, PASCAL has no provisions for external procedures or functions. This would be a serious weakness in large systems (10,000 lines), where the most trivial modification in one of the programs would require the recompilation of the entire system. However, PASCAL does have the CONST feature for declaring program constants, high level data structures and operators, the subrange type, and the control structures for structured programming. All these features make programs easier to read and modify.

(c) reliability

PASCAL performs complete type checking at compile time (including procedure and function parameters, and pointer variables). PASCAL is also a high level and well structured language, so that programs should be smaller and more self-documenting than programs written in languages with fewer data or control structures. It should be considerably easier to write reliable programs in PASCAL than in a language like FORTRAN.

H. Use

PASCAL has been implemented on almost all commercial computer systems, including the PDP 10, PDP-11 series, B6700, UNIVAC 1100 series, IBM 360 and 370 series, and the CDC 3000 and 6000 series. The compiler is written in PASCAL itself, so the compiler could be transported to other machines using standard bootstrapping techniques.

2.12. PREST4

2.12.1. LANGUAGE FEATURES

PREST4 [KAF75] is a preprocessor for Fortran that was developed by a group at Ohio State University during the period 1973-1975. The language provides a number of structured programming constructs, as well as some statements for controlling the output listings of PREST4 source programs. The structured programming constructs are not preceded by special characters (e.g. $, %), a technique that has been used by other Fortran preprocessors. In the remainder of this section, the PREST4 language is considered to be Fortran IV augmented by the PREST4 preprocessor.

A. Basic Data Types and Operators

PREST4 supports the five basic data types of Fortran IV: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL. The language permits mixed-mode expressions and will automatically convert between integer, real, and double precision numbers. Constants used in expressions can have the following types: integer, real, double precision, complex, logical, octal, and character strings (character strings must be delineated by apostrophes, since PREST4 does not permit the H specification used by Fortran IV).

The operators and the data types on which they operate are listed below:

arithmetic operators (INTEGER, REAL, and DOUBLE PRECISION operands)

+, -, *, /, **

logical operators (LOGICAL operands)

.NOT., .AND., .OR.

relational operators

.EQ., .NE.	All types.
.LT., .LE., .GT., .GE.	INTEGER, REAL, or DOUBLE

PRECISION operands only.

B. Control Structures

- IF <expr> THEN <stmt> { ELSE <stmt> }

  (Standard conditional.)

- DO <stmt-list> END

  (Compound statement.)

- DO WHILE <expr>
  UNTIL

  <stmt-list>

  END

  (While and repeat loops.)

- DO <var> = <expr-1> { STEP <expr-2> } WHILE <expr-3>
  UNTIL

  <stmt-list>

  END

  (Standard for loop.)

Note: PREST4 does not permit use of the Fortran IV forms of the IF and DO statements; only the structured forms may be used.

- GOTO <stmt-number>

  GOTO <assign-variable>

  GOTO (<stmt-number-1>, ..., <stmt-number-k>), <var>

  (Unconditional, ASSIGNed, and computed goto statements.)

- READ (<unit-number>, <format-number>, END: <stmt>)
  <input-variable-list>

  (Standard Fortran READ statement with different syntax for the end-of-file condition. The END: <stmt> provides a means of intercepting an end-of-file condition without introducing a GOTO statement.)

- <type> FUNCTION <ident> (<parameter-list>)

  <stmt-list>

  END

SUBROUTINE <ident> { (<parameter-list>) }
<stmt-list>
END

(Standard Fortran function and subroutines. Neither can be recursive. Both functions and subroutines can have multiple entry points.)

C. Data Structures

PREST4 has only one feature for building more complex data types: arrays of up to 7 dimensions. The declaration

DIMENSION <ident> (<dimension-list>)

declares <ident> to be an array. Elements of an array are accessed using standard subscript notation <ident> (<subscript-list>).

D. Other Features

PREST4 is essentially a Fortran language with some additional constructs for structured programming. The language has no block structure or recursion. PREST4 provides statement functions, EQUIVALENCE, COMMON, and DATA statements, and the Fortran I/O statements. Comments are denoted by an asterisk in the first column of the input card. PREST4 also provides a number of control statements for affecting the output listings of a PREST4 program:

%LIST	- Begin listing source program.
%XLIST	- Stop listing source program.
%PAGE	- Page eject.
%SKIP <count>	- Skip specified number of lines.
%DOC	- Places comments in boxes of asterisks.
%DOCEND	

A control statement %COPY <file-name> is also provided for inserting program text into a PREST4 program from a file. This feature would be very useful for inserting variable declarations or COMMON blocks into a program.

E. Runtime Environment

PREST4 has no dynamic storage allocation or recursion, so no stack or heap is needed. Except for I/O and type conversion routines, PREST4 should run on a bare machine.

F. Syntax

Fortran IV (and therefore PREST4) has a BNF grammar, but a compiler would probably not use it. Fortran compilers tend to use ad hoc compiling techniques.

2.12.2. CHARACTERISTICS

A. Machine Dependence

ANSI standard Fortran IV (and therefore PREST4) is fairly machine independent. Fortran programs can usually be transported to different machines with only minor modifications (e.g. different I/O unit numbers).

B. Efficiency

Fortran IV formatted I/O must be performed interpretively and is therefore quite slow. In all other respects Fortran IV and PREST4 are efficent programming languages. We note, however, that the additional structuring of PREST4 programs that would be very helpful to a code optimizer is not available to the Fortran compiler; all the structured statements are converted to IF and GOTO statements before reaching the compiler.

C. Level

PREST4 is a medium level language.

D. Size of Language and Compiler

Because of the EQUIVALENCE statement, the unstructured nature of Fortran programs (optimization is difficult), and the preprocessor pass, PREST4 will require a fairly large compiler.

E. Special System Features

Although PREST4 is described as "A Highly Structured FORTRAN Language for Systems Programming", the language has no special system features.

F. Error Checking and Debugging

Fortran compilers have traditionally had very poor compile and runtime diagnostics, so PREST4 diagnostics will probably be poor. The preprocessor phase of PREST4 does print error messages when illegal PREST4 statements are detected.

PREST4 has two control statements for debugging programs. The statement %IDENT <message> will cause <message> to be printed each time the IDENT statement is encountered during execution of the program. A full statement trace can be initiated with the %TRACE statement.

G. Design Support

(a) modularity

PREST4 supports independent compilation of subroutines and functions, and communication through COMMON blocks.

(b) modifiability

PREST4 has a limited number of structured programming constructs, and an include feature (%COPY) to insert source statements into a PREST4 program from a file. However, the language has no macro processor, no feature like the PASCAL constant statement for declaring program constants, and no significant features for constructing complex data structures. PREST4 programs would be easier to modify than ordinary Fortran IV programs, but more difficult than programs written in languages like PASCAL or HAL/S.

(c) reliability

The structured programming constructs make PREST4 a great improvement over Fortran IV. However, PREST4 has no character or string operators and data types, and does not have sufficient data structuring capabilities. The lack of these features

requires PREST4 programs to simulate any character processing, list processing, or record processing with Fortran code. PREST4 programs will therefore tend to be longer than necessary and more difficult to understand.

H. Use

PREST4 is implemented on the PDP-10, but the preprocessor could be implemented on almost any machine.

2.13. SIMPL-T

2.13.1. LANGUAGE FEATURES

SIMPL-T [BAS74,BAS76a] is a small, procedure oriented, non-block structured language developed by Victor Basili and Joe Turner at the University of Maryland. The language provides features for arithmetic, character, and string processing, and includes a number of structured programming constructs. SIMPL-T is the basis language for a family of languages that includes SIMPL-S and SIMPL-XI (systems programming languages for the Univac 1100 and the DEC PDP-11 series), GRAAL (a graph algorithm language), and SIMPL-R (a language for scientific programming).

A. Basic Data Types and Operators

SIMPL-T has three basic data types: INT (integer), CHAR (single character), and STRING (variable length character strings). Complete type checking is performed at compile-time, and in general no automatic type conversions are performed. SIMPL-T allows six types of constants: integer, character, string, binary, octal, and hexadecimal.

SIMPL-T provides the following operators and functions for manipulating the basic data types:

arithmetic operators (INT operands only)

+, -, *, /, unary -

relational operators (INT, CHAR, or STRING operands)

=, <>, <, >, <=, >=

The operand types must be the same. The relational operators yield an integer result (0 - false, 1 - true).

string operators & functions

<string> .CON. <string>

Concatenation of strings.

<string> [<start-position>, <number-of-chars>]

Substring operator. May appear

on the left-hand side of an assignment statement.

LENGTH (<string>) Current length of string.

MATCH (<string-1>,<string-2>) Position of <string-2> in <string-1>.

INTF (<string>) Converts string to integer.

STRINGF (<integer>) <char> Converts an integer or a character to a string.

TRIM (<string>) Trims trailing blanks.

LETTERS (<string>) Predicate returning true if <string> contains only letters.

DIGITS (<string>) Similar predicate for digits.

CHARF (<string>) Converts from string to character.

logical operators (INT operands)

.AND., .OR., .NOT.

The logical operators all return an integer result (0 or 1).

bit and part-word operators (INT operands)

<integer-expr> .LL. .LC. .RL. .RA. <number-of-bits>

Left logical, left circular, right logical, and right arithmetic shifts.

<int-expr> .A. .V. .X. <int-expr>

Bitwise and, or, and exclusive or.

.C. <int-expr>

Bitwise complement.

<int-expr> [<bit-position>,<number-of-bits>]

Part-word selector. May appear on left-hand side of an assignment statement.

character functions

INTVAL (<char>)

ASCII code for the character.

CHARVAL (<ASCII-code>)

Character corresponding to the ASCII code.

INTF (<char>)

Converts a character (which must be a digit) to an integer.

CHARF (<integer>)
<string>

Converts an integer or a string to character.

PACK (<char-array-variable>,<string-expr>)

UNPACK (<string-expr>,<char-array-variable>)

Conversion between strings and character arrays.

B. Control Structures

- IF <expr> THEN <stmt-list> { ELSE <stmt-list> } END

(Standard conditional with the required terminator END.)

- { !<label>! } WHILE <expr> DO <stmt-list> END

(While loop with optional label. The label may only be referenced by EXIT statements; SIMPL-T has no GOTO statement.)

- CASE <expr> OF

<case-expr-list> <stmt-list>

:

<case-expr-list> <stmt-list>

{ ELSE <stmt-list> }

END

(Case statement. The <expr> is compared sequentially with the values in each <case-expr-list> ; the <stmt-list> whose <case-expr-list> contains the <expr> is executed. The <expr> and <case-expr-list>'s must all be of the same type, but can be INT or CHAR. The <stmt-list> of the optional ELSE clause is executed only if no <case-expr-list> contains the <expr>.)

- PROC <ident> { (<parameter-list>) }

<proc-body>

<type> FUNC <ident> { (<parameter-list>) }
<function-body>

(Procedure and function definition. Both can be recursive, and both can receive their arguments by value or by reference. All scalar parameters are passed by value unless the REF option is specified.)

- EXIT { (<label>) }

(Exit innermost or label while loop.)

- CALL <ident>) { (<argument-list> }

(Call a procedure.)

- <ident> { (<argument-list>) }

(Invoke a function.)

- RETURN

(Return from a procedure.)

- RETURN (<expr>)

(Return from a function with a result.)

- ABORT

(Terminate execution abnormally.)

Note: SIMPL-T provides no GOTO statement.

C. Data Structures

The only data structure supported by SIMPL-T is the one-dimensional array. The declaration

<type> ARRAY <ident> (<number-of-elements>)

declares <ident> to be a one-dimensional array of the specifed type. The type can be any of the three basic types (INT, CHAR, or STRING). Array elements are referenced using standard subscript notation:

<ident> (<subscript-list>) .

D. Other Features

SIMPL-T has a parameterized macro facility of the form

```
DEFINE  <ident> = <define-string>
```

where the <define-string> is any character string. Parameters in the string are denoted by &n, where n is any integer between 1 and 9. For example:

```
DEFINE NUL = 'CHARVAL(0)',      /* control characters */
       LF = 'CHARVAL(10)',
       MOD = '&1-(&1/&2)*&2'    /* mod function */
```

The language also has simple I/O facilities.

E. Runtime Environment

SIMPL-T requires a runtime stack for recursive procedures and functions, and for evaluation of string expressions. However, no dynamic storage allocator is required for arrays or strings.

F. Syntax

The working BNF grammar for SIMPL-T has approximately 150 productions.

2.13.2. CHARACTERISTICS

A. Machine Dependence

SIMPL-T has few machine dependent features and could be implemented on almost any machine.

B. Efficiency

SIMPL-T has no dynamic arrays, and no automatic type conversion. All type checking is performed at compile time, and the default passing mechanism for procedure or function calls is by value. This allows a great deal of work to be done a compile time rather than at execution time. The Univac 1100 series implementation of SIMPL-T generates code that is as efficient as Univac Fortran V.

C. Level of the Language

SIMPL-T is a medium level language.

D. Size of Language and Compiler

The compiler for SIMPL-T is moderate in size.

E. Special System Features

SIMPL-T has no special system features, although the two system programming languages in the SIMPL family (SIMPL-S and SIMPL-XI) provide a number of system features. SIMPL-XI [HAM76], for example, provides indirect and absolute addressing, access to machine registers, and interrupt procedures (procedures activated when a specific interrupt occurs). A one-dimensional array MEM is used to provide the absolute addressing feature: MEM(I) accesses the I-th word in main memory. A similar array MEMB is provided for accessing bytes.

F. Error Checking and Debugging

SIMPL-T performs complete type-checking at compile time, and no implicit conversion between data types is permitted. SIMPL-T can therefore detect many errors at compile time that can not be detected by other languages (such as Fortran, LITTLE, or BLISS).

A number of compiler directives are also available for debugging SIMPL-T programs, including:

(1) subscript checking
(2) case statement checking
(3) calling history
(4) static and runtime statistics
    (Such as number of statements executed, timing estimates for procedures, and so forth.)
(5) value tracing for program variables
(6) compilable comments
    (The form of a compilable comment is
    /+ <indicators> <SIMPL-text> +/
    where an <indicator> is an integer that can be turned on or off with other compiler directives. For example;

```
/* 4 WRITE ("DEBUG:    ON ITERATION",I,"X =",X) */
```

(7) cross reference and attribute listings.

G. Design Support

(a) modularity

SIMPL-T allows independent compilation of program modules, and communication through external variables and entry points.

(b) modifiability

SIMPL-T has a fairly complete set of structured programming constructs and a powerful macroprocessor. SIMPL-T programs should be fairly easy to modify.

(c) reliability

The lack of constructs for building more complex data structures may make SIMPL-T programs longer than necessary and difficult to read. SIMPL-T has no record structure, and arrays can only have one dimension. A large portion of a SIMPL-T program that operates on complex data structures will therefore be taken up by segments of SIMPL statements providing access methods for the data structures. Languages with more complex data types would provide these access methods automatically. In HAL/S, for example, if A and B are compatible record structures then the statement A = B; will copy all of record B into record A. In SIMPL-T a transfer of this type would have to be simulated by a number of assignment statements.

H. Use

SIMPL-T has been implemented on the Univac 1100 series, the PDP 11/45, the Data General NOVA, and the CDC 6600. A version for the IBM 360 series is under development. The compiler is written in SIMPL-T itself, so the compiler can be transported to other machines using standard bootstrapping techniques. In fact, the same front end (scanner and parser) is used on all implementations of SIMPL-T. This provides standard error diagnostics for incorrect programs.

2.14. SPL / Mark IV

2.14.1. LANGUAGE FEATURES

SPL [SDC70] is a large, high level language developed by System Development Corporation in the period 1967-1970. The language was designed for aerospace applications and combines many of the features in the PL/I and JOVIAL languages. SPL offers high level features like data tables and matrix arithmetic, as well as low level, machine-oriented features like inline assembly language and access to machine registers. SPL has five application oriented subsets; the subset chosen for this report (SPL / Mark IV) was designed for ground-based support computers. In the remainder of this section SPL / Mark IV will be referred to as SPL.

A. Basic Data Types and Operators

SPL has nine basic data types: INTEGER, FIXED, FLOATING, BOOLEAN, LOGICAL (bit string), TEXT (character string), STATUS (ordered sets of "states"), LOCATION (typed pointers), and CONTEXTUAL (a "universal" type). The STATUS type is equivalent to the PASCAL scalar type. CONTEXTUAL items can be assigned a value of any type. When a CONTEXTUAL item X is assigned a value of type T, the type of the item X is assumed to be T until X is assigned a new value of different type on some subsequent line in the program. CONTEXTUAL items are intended to be used for temporary storage of various types of items.

The following types of constants can appear in an SPL expression: integer, fixed point, floating point, boolean, binary, octal, hexadecimal, and character string, location, and status. Mixed mode expressions are permitted and automatic conversion is provided between all of the basic data types.

The operators and the data types on which they operate are listed below:

arithmetic operators

+, -, *, /, **

REM - Remainder function.

ABS - Absolute value.

LSH - Left arithmetic shift.

RSH - Right arithmetic shift.

SCL - Scales an arithmetic expression.

SCLR - Scales and rounds an arithmetic expression.

logical operators

LAND, LOR, LXOR, LSH, RSH

- Bitwise and, or, exclusive or, and left and right logical shift.

BIT - Pseudo-variable for accessing bit strings in any type of item. Can appear on left-hand side of an assignment statement.

boolean operators

NOT, AND, OR, EQUIV

- All the boolean operators yield a boolean result.

relational operators

EQ, NQ, GR, LS, GQ, LQ

- The relational operators yield a boolean result.

character operators

BYTE - Pseudo-variable for accessing bytes in a textual item. Can appear on left-hand side of an assignment statement.

location operators

LOC - Yields location of an item.

IND - Pseudo-variable for performing indirect addressing. Can appear on left-hand side of an assignment statement.

B. Control Structures

- IF <boolean-expr> <stmt-list>

(Simple conditional statement.)

- IF <boolean-expr> THEN <stmt-list>
  { ORIF <boolean-expr> <stmt-list> }
  . . .
  { ORIF <boolean-expr> <stmt-list> }
  { ELSE <stmt-list> }
  END

  (Conditional statement. If the initial boolean expression is false then the boolean expressions in the ORIF clauses are evaluated in order until a true one is found. If all the boolean expressions are false then the ELSE statement is executed.)

- CONDITIONS
  <boolean-expr> <indicator-list>
  . . .
  <boolean-expr> <indicator-list>
  ACTIONS
  <stmt> <indicator-list>
  . . .
  <stmt> <indicator-list>
  ELSE <stmt>
  END

  (Decision table for creating a tabular solution to a complex decision problem. The table describes the conditions applicable to the problem and the actions to be taken in response to the conditions. The indicator lists in the CONDITIONS and ACTIONS sections are composed of indicators Y (yes), N (no), and blank (doesn't apply). The indicator N can only be used in the CONDITIONS section.)

- FOR <var> { = <init-value> } { BY <incr-expr> }
  { WHILE <boolean-expr> } { UNTIL <boolean-expr> }
  <numeric-expr>
  <stmt-list> END

(For loop. If the <init-value> clause is not specified then the current value of the loop variable is used, and if the BY clause is not specified the loop variable is not automatically incremented on each iteration of the loop. The value of the loop variable and the <incr-expr> can be altered by the loop body. The clause UNTIL <numeric-expr> is equivalent to UNTIL <var> EQ <numeric-expr>.

An abbreviated form of the FOR loop is provided for processing tables (discussed in section D. Data Structures). The statement

FOR <var> = <table-name> <stmt-list> END

is equivalent to

FOR <var> = <length-of-table> - 1 BY -1
WHILE <var> GQ 0 <stmt-list> END .)

- FOR <var> = <for-clause>
  ALSO <var> = <for-clause>
  . . .
  ALSO <var> = <for-clause>
  <stmt-list>
  END

(Parallel FOR loop. All of the loop variables are incremented on each iteration of the loop. The <for-clause> contains the <init-value>, BY, WHILE, and UNTIL clauses of the ordinary FOR statement.)

- LOOP WHILE <boolean-expr> <stmt-list> END
  UNTIL

(While and until loop with the test performed before execution of the loop body.)

- ON <boolean-expr> <stmt-list> END
  <interrupt-name>

(Feature for handling abnormal conditions. The <boolean-expr> is automatically evaluated whenever the first operand in the expression (which must be a

variable) is assigned a new value. The variable can be assigned a new value by an SPL statement or by some hardware event. If the <boolean-expr> evaluates to true or if the specified interrupt occurs then the <stmt-list> is executed . The SPL ON statement is similar to PL/I ON-conditions, although SPL provides no way of selectively enabling or disabling ON variables, or for changing the <stmt-list> to be executed for a given condition or interrupt.)

- UNLOCK <interrupt-name>
  LOCK

  (Enables or disables the specified interrupt. The LOCK and UNLOCK statements can also be used for reserving hardware registers.)

- GOTO <label>
  <location-variable>

  (Unconditional transfer. The location variable is assumed to contain the address of some SPL statement label.)

- GOTO <switch-name> (<integer-expr>)

  (Computed goto. The <switch-name> must have been declared with a statement of the form

  SWITCH <switch-name> = <label-list>

  On execution of the GOTO statement the value i of the <integer-expr> is used to select the i-th label, and a branch is made to the selected label.)

- RETURN { (<result-expr>) }

  (Return from a procedure or function with an optional result.)

- STOP { (<label>) }

  (Halts execution. A "continue operation" after the execution of a STOP statement will result in a transfer of control to the statement following the STOP statement, or to the specified label.)

- TEST { (<FOR-loop-variable>) }

  (Continues the next iteration of the innermost WHILE, UNTIL, or FOR loop, or the innermost FOR loop having the specified loop variable.)

- WAIT

  (Repeats execution of an IF, WHILE, or UNTIL statement until the conditional expression is satisfied. The statement is intended to be used to halt the execution of a program until some external event has occurred. For example,

  IF STATUSREG EQ X'2C' THEN WAIT .)

- PROC .<proc-name> { (<input-parameter-list>
  = <output-parameter-list>) }

  { <type> } { INLINE / REENTRANT / RECURSIVE }

  <data-declarations>
  ENDDATA
  <stmt-list>
  EXIT { (<result-expr>) }

  (Procedure or function definition. Procedures, functions, and statement labels can be passed as procedure parameters. Alternate exits from a procedure are possible by branching to a statement label parameter. Both procedures and functions can have multiple entry points. The EXIT clause at the end of a function definition indicates the expression to be returned by the function, although the EXIT expression can be overridden by a RETURN statement.)

- .<proc-name> { (<input-arguments> = <output-arguments>) }

  (Invoke a procedure or function.)

- CLOSE <close-name> <stmt-list> END

  (Parameterless, internal subroutine that can be defined within another procedure.)

- GOTO <close-name>

  (Call a CLOSE subroutine. Control will resume at the next statement when the CLOSE routine has finished execution.)

C. Data Structures

SPL has three features for constructing more complex data structures from the basic data types:

(a) arrays

Arrays are declared with a statement of the form

ARRAY <ident> (<dimension-list>) <type> { MEDIUM } DENSE

The <type> can be any of the basic data types, and the options MEDIUM and DENSE affect the packing density of the array. Arrays can have an arbitrary number of dimensions. The <dimension-list> can optionally contain implicit subscripts for each of the dimensions. These implicit subscripts are used as the default subscripts whenever an array variable is used without explicit subscripts. For example:

```
ARRAY M(I 10, J 10) INTEGER  " Matrix with implicit "
FOR I = 0 BY 1 UNTIL 10      " subscripts.          "
ALSO J = 0 BY 1
  M = I                      " Equivalent to        "
END                          " M(I,J) = I .         "
```

Array indexing begins at 0, and array elements are referenced using the standard subscript operator <ident> (<subscript-list>) . The following operators are available for manipulating arrays, matrices (2 dimensional arrays or 2 dimensional subsets of arrays), and vectors (rows or columns of matrices):

=	- Assignment.
==	- Exchange.
+, -	- Vector and matrix addition.
*	- Vector and matrix dot product.

**-1 - Matrix inverse.
TPOSE - Matrix transpose function.
/* - Cross product for 3-D vectors.

(b) tables

SPL has a table data structures almost identical to the JOVIAL/J3B table. Tables are declared with a statement of the form

TABLE <ident> { (<implicit-subscript>) } <table-length>
{ SERIAL } { MEDIUM / DENSE / TIGHT } <item-declarations>

The implicit subscript is used whenever the table identifier is used a subscript. The default method for allocating tables is by "columns", that is, there is a contiguous block of core for the first item in all table entries, another block for all the second items, and so forth. If the SERIAL option is specified, however, the table will be allocated by table entry. For each table entry there will be a block of core long enough to contain all the items in the entry. The options MEDIUM, DENSE, and TIGHT affect the packing density of the table. The items in the <item-declarations> list can be any of the basic data types, but item names must be distinct between tables.

An alternate version of the table declaration gives the programmer complete control over placement of items within a table entry. The number of words per table entry and the placement of each item (word position and starting bit within the word) is directly specified. The storage for items can overlap.

Tables can be accessed in any of the following four ways:

<table-name> - Accesses entire table.
<table-name> (<subscript>) - Accesses all of the specified entry in the table.
<item-name> - Accesses an entire column of the table.
<item-name> (<subscript>) - Accesses a single item in the

specified table entry.

The assignment operator =, the exchange operator ==, and the relational operators EQ, NQ can be used to copy, exchange, or compare tables or table entries. The assignment operator can also be used to copy columns of a table. Finally, the functions NENT and NWDSEN are provided for determining the number of entries in a table and the number of words in a table entry.

(c) record structures

Record structures are declared with the statement

<ident>. DECLARE <member-declarations>

The members in a record can be arrays, tables, or any of the basic data types. The identifiers used for members need not be distinct from identifiers declared elsewhere. The ' operator is used to access members in a record:

<record-name> ' <member-name>

D. Other Features

SPL has an OVERLAY statement that is equivalent to the Fortran EQUIVALENCE statement, and extensive facilities for sequential I/O (including programmer specified blocking factors, record format, error exits, data conversion, and statements for opening and closing files).

Simple replacement macros can be defined using the DEFINE statement

DEFINE <ident> AS <character-string>

All occurrences of the identifier are replaced by the character string. SPL also has a CONSTANT declaration for declaring program constants, and an implementation dependent COMPOOL feature that is similar to the JOVIAL COMPOOL file.

The language provides default declarations for undeclared variables, and the programmer can change the default to any of the basic data types at any point in an SPL program. Finally, the SDC implementation of SPL / Mark IV has compiler directives permitting the user to write portions of an SPL program in the

JOVIAL language.

E. Runtime Environment

SPL requires a runtime stack for programs using recursive or reentrant procedures and functions, and sequential I/O routines.

F. Syntax

The BNF grammar for SPL has approximately 400 productions.

2.14.2. CHARACTERISTICS

A. Machine Dependence

SPL has a large number of machine dependent features, including the function BIT for accessing bit strings, the OVERLAY statement, user specified table allocation (word position and bit position within a word), the hardware statement, and inline assembly language.

B. Efficiency

SPL permits efficient programming. The language provides high level operators (including matrix arithmetic and direct assignment of arrays and tables), a structured control structure that permits better optimization, many features for minimizing storage requirements (the OVERLAY statement, user defined tables, packing densities), the INDEX statement for frequently accessed variables, and the ability to generate inline assembly code. No runtime stack is required for non-reentrant procedures.

C. Level of the Language

SPL is a high level language, although it also provides a large number of low level features.

D. Size of the Language and Compiler

SPL ia a large language and will require a large compiler.

E. Special System Features

SPL has many features that would be helpful in systems programming, including

(a) Pointers, tables, and record structures.

(b) Recursive, reentrant, or inline procedures. Procedures can have multiple entry points and alternate exits. A programs can abort to a procedure many "levels" back up the calling chain by branching to a statement label passed as an input parameter.

(d) The OVERLAY statement, and user defined table allocation permitting the overlaying of data items and access to a block of core under varying data formats.

(e) The ON statement for intercepting interrupts and abnormal conditions.

(f) The HARDWARE statement for defining machine registers and other hardware, and the DIRECT statement for inline assembly language.

(g) The LOCK and UNLOCK statements for reserving hardware registers, enabling and disabling interrupts, and establishing read/write protection for areas of memory (for machines having a memory protection facility).

(h) The INDEX statement for requesting that frequently accessed variables be allocated in the fastest storage locations available.

F. Error Checking and Debugging

In general, SPL requires careful programming. Automatic conversion is peformed bewteen the basic types, and default declarations are provided. This will tend to hide a number of programming errors such as misspellings. Location variables can be used to alter instructions or to branch into a data area. Finally, the language has many system features the permit the user to directly access hardware facilities.

SPL has two compiler directives that would be helpful in debugging and improving SPL programs. The TRACE directive is used to trace the value of selected program variables and the flow history of statement labels for selected areas in a program.

The TIME directive enables the programmer to determine the execution time of any block of SPL statements.

G. Design Support

(a) modularity

SPL is quite modular. The language has internal and external procedures and functions, the CLOSE routine for nested procedures, a structured control structure, and the COMPOOL file. Independent compilation of procedures and functions is permitted.

(b) modifiability

The language has a variety of basic data types, high level operators, the CONSTANT attribute for declaring program constants, the DEFINE statement for declaring simple macros, and a structured control structure. All of these features would make SPL programs easier to read and modify.

However, SPL also has many machine dependent features that permit bit packing, overlaying of data areas, and inline assembly language. Use of these features in a program would make modification or transportation to other machines difficult. The language also permits programs to be written that are not "self documenting". Implicit subscripts are provided for arrays and tables, automatic type conversions are performed, and default declarations are provided for undeclared variables. The statement GOTO A in an SPL program can be an unconditional branch to the statement labeled A or a call of a parameterless procedure.

(c) reliability

SPL provides many low level features that permit efficient, machine dependent programming at the expense of reliability. All of the system features require careful programming. Automatic type conversions and default declarations will also tend to hide program errors.

H. Use

SPL has been implemented on the IBM 360 and 370 series and on the CDC 6000 series. The compiler was developed by SDC using the translator writing system CWS.

2.15. STRCMACS

2.15.1. LANGUAGE FEATURES

STRCMACS [BAR74] is a set of macros providing structured programming constructs for IBM OS/360 assembly language. The macros, which were developed by C. Wrandle Barth at Goddard Space Flight Center, are placed in the OS/360 macro library and invoked automatically during the assembly of an STRCMACS program. No preprocessor step is required. In the remainder of the section, STRCMACS will be considered to be the structured programming macros plus all the facilities of OS/360 assembly language.

A. Basic Data Types and Operators

STRCMACS is a macro assembly language operating on 32-bit words, and no type checking is performed. The operators are the OS/360 assembly language instructions. The instruction set provides instructions for manipulating bits, characters, integers, and floating point, double precision, extended precision, and decimal numbers.

B. Control Structures

- BLOCK
  <instruction-1>
  .
  .
  .
  <instruction-k>
  BLEND
  (Compound statement or code block.)
- IF <test expression>
  <instruction list>
  { ELSE
  <instruction list> }
  FI
  (Standard conditional statement. The <test expression> is composed of machine instructions for setting the condition code and mnemonics (from the extended

branch-on-condition mnemonics) specifying under what conditions the "then" part of the if statement is to be executed. Tests in the test expression may be combined using the connectives AND and OR. For example:

```
IF (LTR,3,3,P)                    - if register 3 > 0
IF (TM,8(1),X'80',0),OR,(CLC,SIZE,=C'MAX ',EQ)
                                  - if high order bit of
                                    word at 8(1) is set,
                                    or if SIZE = 'MAX '
```

```
- DO FOREVER
    <instruction-list>
  OD
```

(Unbounded repetition.)

```
- DO WHILE, <test expression>
     UNTIL
    <instruction list>
  OD
```

(Standard while and repeat loops. The <test expression> is identical to the one described for the IF construct.)

```
- DOCASE <case var>
    CASE <instruction list> ESAC
     .
     .
     .
    CASE <instruction list> ESAC
  ESACOD

  DOCASE <case var>
    CASE <case value list> <instruction list> ESAC
     .
     .
     .
    CASE <case value list> <instruction list> ESAC
  ESACOD

  DOCASE
    CASE <test expression> <instruction list> ESAC
     .
     .
     .
```

CASE <test expression> <instruction list> ESAC

ESACOD

(Case and Select constructs. In the first two forms the <case var> is used to select one of the CASE blocks for execution. The <case var> can be a register or a memory location, and can be a byte, halfword, or fullword in length. In the first form of the DOCASE the <case var> is used directly to select a CASE block (if <case var> = i then the i-th CASE block is executed). In the second form, the <case var> is compared sequentially with the <case value list>s; the CASE block whose <case value list> contains the <case var> is executed. In the third form of the DOCASE there is no <case var>; the <test expressions> are executed sequentially until one of the tests succeeds. The CASE block containing the succeeding test is then executed.

In any of the three forms of the DOCASE construct one of the CASE blocks can have the MISC operand (for miscellaneous). A CASE block with this attribute is executed if no other CASE blocks in the DOCASE are executed.

example:

case block	block will be executed if
CASE 2	<case var> = 2
CASE 3,(5,12)	<case var> = 3,5,6,...,12
CASE '=','<>'	<case var> = '=' or '<>'
CASE ('I1','I5')	<case var> = 'I1',...,'I5'
CASE (LTR,8,8,Z)	register 8 = 0)

- <label> PROC (<options>)
  <instruction list>
  CORP <label>

(Procedure construct. The procedure can be called using the OS/360 CALL macro. The <options> operand allows the user to specify standard or non-standard linkage, dynamic saveareas, a procedure identifier

string, base registers, and so forth.)

- EXIT <label>

  (Exit the specified block.)

- ONEXIT

  <instruction list>

  [ATEND

  <instruction list>]

  OD

  (STRCMACS distinguishes between normal exit of a DO loop by failure of the loop test and abnormal exit by the execution of an EXIT macro. The ONEXIT ... ATEND construct can be appended to any of the DO loop constructs. If the loop is terminated abnormally then the ONEXIT <instruction list> is executed and the ATEND <instruction list> is skipped. IF the loop terminates normally only the ATEND <instruction list> is executed.)

C. Data Structures

STRCMACS has no high level data structures.

D. Other Features

An STRCMACS program can use all of the OS/360 assembly language instructions.

E. Runtime Environment

As an assembly language STRCMACS will run on a bare machine.

F. Syntax

The STRCMACS macros are translated into assembly language by the OS/360 assembler. There is no compiler for STRCMACS.

2.15.2. CHARACTERISTICS

A. Machine Dependence

The STRCMACS macros are designed for the IBM 360 series. However, similar macros could be designed for any machine.

B. Efficiency

STRCMACS is as efficient as assembly language.

C. Level

STRCMACS is a very low level language.

D. Size of Language and Compiler

STRCMACS is implemented by a small number of macros, and is therefore quite small.

E. Special System Features

STRCMACS has no special constructs or data structures for systems programming. However, the user has access to the full set of OS/360 assembly language instructions.

F. Error Checking and Debugging

The STRCMACS macros will produce diagnostic messages at assembly time if an error is detected. However, no runtime error checking is performed. A few features are provided for debugging STRCMACS programs. Any PROC can specify the following debug options: (1) LISTBLOCKS - lists the static nesting, name, and block number of all blocks in the PROC; (2) PROCNAMES - generates an in-line character string for the procedure name to aid in locating procedures in an ABEND dump; (3) PROCCOUNTS, BLOCKCOUNTS - counts the number of times that each block in the PROC is executed; (4) PROCTRACE - maintains the calling history of the last 257 blocks.

G. Design Support

(a) modularity

STRCMACS supports independent assembly of programs, and provides communication through external variables or COMMON blocks. The language is also considerably more structured than

ordinary assembly language.

(b) modifiability

STRCMACS is essentially an assembly language. Although the structured programming constructs are a vast improvement over ordinary assembly language, STRCMACS programs will still be difficult to modify.

(c) reliability

Although the structured constructs are an improvement, STRCMACS will still have the same reliability problems as assembly language. No type checking of any sort is performed, all the operators (machine instructions) are low level, and there are no data structuring facilities.

H. Use

STRCMACS is implemented on the IBM 360 series. Since the structured programming constructs are not machine dependent, and since the number of macros is small, STRCMACS could be implemented on other machines without any significant effort.

3. POCCNET REQUIREMENTS

In this chapter we examine the specific requirements of POCCNET [DES76a,DES76b] and its applications software. POCCNET is a hardware/software system that will support the development and operation of Payload Operations Control Centers (POCCs) during the 1980's. In order to implement the POCCNET system, software must be developed for the distributed computer network and the standardized applications software. We will therefore give a brief description of each of these areas.

The POCCNET network is composed of five functional subsystems: an Applications Processor (AP) subsystem, an Interprocess Communication (IPC) subsystem, a Data Base (DB) subsystem, an Interface subsystem, and a Control subsystem. The AP subsystem is composed of general purpose minicomputers with operating systems capable of running POCC software. The IPC subsystem handles all message transfers within the network, and the DB subsystem provides on-line storage for the POCCs and the network. The standardized applications software for POCCNET is also managed by the DB subsystem. The Interface subsystem provides communication between POCCNET and the outside world (which includes human users, telemetry and commands, and other computer systems). Finally, the Control subsystem directs and monitors the operation of the entire POCCNET system.

The package of standardized applications software will provide software that implements functions common to many POCCs. This includes POCC application programs, program development tools, and related software.

The implementation language (or group of languages) for POCCNET will therefore have to support all of the following application areas: (1) general systems programming, which is required throughout POCCNET; (2) real-time processing for time-critical operations in the IPC and Interface subsystems; (3) data-base processing for the DB subsystem; (4) numerical processing for massaging spacecraft data, simulating telemetry, and so forth; (5) data formatting and conversion for the

Interface subsystem. This involves primarily bit and character string processing.

For each of these application areas we would like a programming language that provided the following features:

(1) general systems programming
   (a) bit and character string manipulation
   (b) some ability to perform absolute and indirect addressing (such as pointers or the SIMPL-XI MEM array)
   (c) record structures and one-dimensional arrays
   (d) ability to suppress type checking, so that a block of core can be accessed under various data formats
   (e) exception handling by ON-conditions or interrupt procedures
   (f) reentrant or recursive procedures and functions
   (g) dynamic storage allocation
   (h) concurrent processes and controlled data sharing between processes
   (i) access to operating system facilities

(2) real-time processing
   (a) all of the features of general systems programming
   (b) high efficiency
   (c) real-time scheduling of processes (schedule at a certain time, in a certain number of clock ticks, and so forth)

(3) data-base processing
   (a) protection mechanism for files and individual data elements
   (b) various file organizations and access methods
   (c) good data structuring capabilities, possibly a data abstraction feature
   (d) facilities for defining a data-base management system

(4) numerical processing
   (a) variety of arithmetic data types and precisions
   (b) user control over precision
   (c) ability to intercept underflow and overflow conditions
   (d) array, matrix, and vector data structures

(e) library of mathematical functions and subroutines

(5) data formatting and conversion

(a) bit and character string manipulation

In addition to the requirements for the separate application areas, there are a group of features that should appear in any POCCNET implementation language. These features include integer, floating point, and character data types; control structures for structured programming; arrays and record structures for building data structures; a macro processor; and some form of INCLUDE statement for copying commonly used source files into a program. A data abstraction facility would also be very helpful, although CS-4 and Concurrent Pascal are the only languages in this study that provide such a feature.

In addition to having all of the above capabilities, the scientific programming notation should possess certain characteristics. Among other things it should support ease of program expression, the writing of correct, efficient, and portable code, and the reuse of algotithms written in it. Let us consider these characteristics one at a time.

One would like to express the algorithms in a natural manner. This implies the notation should be natural to the problem area. For example within the general problem area of mathematics there is a specialized and different mathematical notation for the algebraist and analyst. Each aids in expressing the problems of the particular area explicitly and precisely and in an easy to communicate form.

Correctness of a program is defined as the ability of the program to perform consistantly with what we perceive to be its functional specifications. The programming language should support the writing of correct programs. The language should simplify rather than complicate the understanding of the problem solution. The complexity in understanding a program should be due to the complexity inherent in the algorithms, not due to the notation used. The notation should be clear and simple. A language natural to the problem area aids in correctness as it

makes the statement of the solution easier to read and understand. The easier it is to read and understand a solution algorithm, the easier it is to certify its correctness. Aids in making a program readable are structuring it from top to bottom and breaking it into small pieces. In order to achieve the goal of supporting correctness, a language should be simple, contain well-understood control and data structures, permit the breaking up of the algorithm into small pieces using procedures and macros, and contain high-level problem area oriented language primitives.

A program is considered efficient if it executes at as fast a speed and in as small a space as is necessary. The language should permit the efficient execution of programs written in it. The higher level the algorithm, the more information is exposed for optimization and the better job a compiler can do on improving the code generated. On the other hand, high level often implies general applicability in order to handle the majority of cases. This can often imply an inefficiency for a particular application. For example, consider a language in which matrices have been defined as a primitive data type with a full set of operators including matrix multiplication. The multiplication operation has been defined for the general case. Suppose the particular subproblem calls for the multiplication of two triangular matrices. Using the standard built in operator is inefficient. One would like to be able to substitute a more efficient multiplication algorithm for the particular case involved. But this implies that the language permits the redefinition of language primitives at lower levels of abstraction. That is, the programmer should be able to express the algorithm at a high level and then alter the lower level design of the algorithm primitives for a particular application when it is necessary for reasons of efficiency.

A language supports portability when it permits the writing of algorithms that can execute on different machines. Portability is a difficult, subtle problem that involves several diverse subproblems. The numerical accuracy of arithmetic

computations can vary even on machines with the same word size. Techniques for dealing with this problem include variable length arithmetic packages or a minimum precision (modulo word size) specifications. Another problem area of portability is text processing. One way of dealing with this problem is to define a high-level string data type which is word size independent. A third area of problems involves interfacing with a variety of host machine systems. One method of handling this is to define programs to run on some level of virtual machine that is acceptable accross the various machine architectures and systems and then to define that virtual machine on top of the host system for each of those architectures. This is commonly done using a runtime library. In general the higher level the algorithms, the more portable they are. However, more portability often means less efficiency. A language that supports portability should contain one of the above mechanisms for transporting numerical precision accross machine architectures, high level data types, the ability to keep nonportable aspects in one place, and a macro facility for parameterizing packets of information modulo word size.

Software is reusable if it can be used accross several different projects with similar benefits. In order for software to be reusable, its function must be of a reasonably general nature, e.g., the square root and sine functions, it must be written in a general way and it must have a good, simple, straightfoward set of specifications. The area of scientific programming has a better history of reusable software than most. Consider as examples some of the libraries of numerical analysis routines. This is due largely to the easily recognizable, general nature of many scientific functions and the simplicity of their specifications. However, there are whole areas of scientific software development that do not have a histroy of reuse, such as telemetry software.

Software written in a general way may perform less efficiently than hand-tailored software. However, if it is well written it should be possible to measure it and based on these

measures modify it slightly in the appropriate places to perform to specification for the particular application.

A good, simple, straightfoward set of specifications is not easy to accomplish, especially when the nature of the function is complex. A good high level algorithm can help in eliciting that specification. Specifications for software modules should also include an analysis of the algorithm, e.g., the efficiency of the algorithm with respect to the size of the input data. The language should support the development of a good library of well-specified software modules that are easy to modify if the time and space requirements are off. It should also be capable of interfacing efficiently with other languages and of expressing algorithms so that the essential function is clear and of a general nature.

4. LANGUAGE FEATURE TABLES FOR THE LANGUAGES

4.1. INTRODUCTION

Chapter 2 contained a discussion of the criteria used for evaluating the fifteen languages, and the preliminary evaluations of the languages themselves. This chapter contains a series of tables that summarize the evaluations in Chapter 2, as well as adding some new information about the languages and POCCNET requirements discussed in Chapter 3. Each table is devoted to one of the following POCCNET requirements: modularity, modifiability, reliability, data structuring, character string processing, bit string processing, numerical processing, efficiency, special system features, and error checking and debugging. Each table contains the primary language features that influence the POCCNET requirement, and indicates for each language feature the presence (X) or absence (.) of that feature in the languages. Footnotes are added to the end of some of the tables to provide additional information about a language or language feature.

The following abbreviations are used for the languages in the tables: BL (BLISS-11), C (C), CP (CONCURRENT PASCAL), FL (FLECS), HS (HAL/S), IF (INTERDATA FORTRAN V), JS (JOSSLE), JV (JOVIAL/J3B), LI (LITTLE), PA (PASCAL), P4 (PREST4), SI (SIMPL-T), and SP (SPL / Mark IV). The language STRCMACS is not included in the tables because it only provides structured control structures (no data types or data structures).

4.2. MODULARITY

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Structured control structure	X	X	X	X	X	X	.	X	X	X	X	X	X	X
Independent compilation of programs	X	X	.	X	X	X	X	X	X	X	.	X	X	X
INCLUDE feature [1]	X	X	.	.	.	.	.	.	X	.	.	X	.	.
COMPOOL files	.	.	.	.	.	X	.	.	X	.	.	.	.	X
Global or COMMON data	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Controlled access to shared data [2]	.	.	X	X	.	X	.	.	.	.	.	.	.	.
Data abstraction facility	.	.	X	X	.	.	.	.	.	.	.	.	.	.
Block structure [3]	X	.	X	X	.	X	.	X	X	.	X	.	.	X

Notes:

[1] Some feature permitting source text from a program library to be included into a program.

[2] Such as the HAL/S UPDATE block.

[3] JOSSLE is a block structured language, but it restricts the inheritance of global variables. See discussion of KNOWN statement in the Chapter 2 evaluation of JOSSLE.

4.3. MODIFIABILITY

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Structured control structure	X	X	X	X	X	X	.	X	X	X	X	X	X	X
INCLUDE feature [1]	X	X	.	.	.	.	.	.	X	.	.	X	.	.
COMPOOL files	.	.	.	.	.	X	.	.	X	.	.	.	.	X
Data abstraction facility	.	.	X	X	.	.	.	.	.	.	.	.	.	.
Simple replacement macros [2]	X	X	.	.	.	X	.	.	X	X	.	.	X	X
Parameterized macros	X	.	.	.	.	X	.	.	X	X	.	.	X	.
Conditional compilation of source text	.	.	.	.	.	.	X	.	X	.	.	.	X	.
High level data structures and operators	.	.	X	X	.	X	.	X	X	.	X	.	.	X
CONSTANT declaration	.	.	X	X	.	X	.	X	X	.	X	.	.	X

Notes:

[1] Some feature permitting source text from a program library to be included into a program.

[2] Macros that do not permit parameters.

4.4. RELIABILITY

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Structured control structure	X	X	X	X	X	X	.	X	X	X	X	X	X	X
Full type checking [1]	.	X	X	X	.	X	.	X	.	X	X	X	X	.
INCLUDE feature [2]	X	X	.	.	.	.	.	.	X	.	.	X	.	.
Data abstraction facility	.	.	X	X	.	.	.	.	.	.	.	.	.	.
COMPOOL files	.	.	.	.	.	X	.	.	X	.	.	.	.	X
High level data structures and operators	.	.	X	X	.	X	.	X	X	.	X	.	.	X
CONSTANT declaration	.	.	X	X	.	X	.	X	X	.	X	.	.	X
Few machine-dependent features	.	X	X	X	X	X	X	X	.	X	X	X	X	.
Standardized output listings	.	.	.	.	X	X	.	.	.	.	.	.	.	.
Debugging aids [3]	.	.	.	X	.	.	X	X	.	X	.	X	X	X
Few compiler-supplied defaults	.	.	X	X	.	.	.	X	.	.	X	.	X	.

Notes:

[1] Including type checking of procedure parameters.

[2] Some feature permitting source text from a program library to be included into a program.

[3] LITTLE and SIMPL-T provide many debugging aids, the other languages provide only a few.

4.5. DATA STRUCTURING FEATURES

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Array data structure	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Array assignment operator	.	.	X	X	.	X	.	X	.	.	X	.	.	X
Array comparison operators = and ~=	.	.	X	X	.	X	.	.	.	.	.	.	.	.
Record data structure [1]	.	X	X	X	.	X	.	X	X	.	X	.	.	X
Record assignment operator	.	.	X	X	.	X	.	X	X	.	X	.	.	.
Record comparison operators = and ~=	.	.	X	X	.	X	.	.	X	.	X	.	.	.
Untyped pointer variables [2]	X	.	.	.	.	.	X	.	.	.	.	.	.	.
Typed pointer variables [3]	.	X	.	.	.	X	.	X	.	.	X	.	.	X
Address function for pointers	X	X	.	.	.	X	X	X	.	.	.	.	.	X
Dynamic storage allocation using pointers [4]	.	.	.	.	.	.	.	X	.	.	X	.	.	.
Set data type	.	.	X	X	.	.	.	.	.	.	X	.	.	.
Set assignment operator	.	.	X	X	.	.	.	.	.	.	X	.	.	.
Set relational operators = and ~=	.	.	X	X	.	.	.	.	.	.	X	.	.	.
Various set operators [5]	.	.	X	X	.	.	.	.	.	.	X	.	.	.
Data abstraction facility	.	.	X	X	.	.	.	.	.	.	.	.	.	.

Notes:

[1] JOVIAL only has a table data structure (tables can only be formed from simple items, so that a JOVIAL table can not contain another table or an array as one of its items).

[2] Pointers in INTERDATA FORTRAN V can only be used to fetch data indirectly, they can not be used to store data indirectly.

[3] The JOSSLE pointer type is really a table index (subscript) and not a general pointer.

[4] Such as the JOSSLE ALLOCATE statement or the PASCAL function NEW.

[5] Such as set union, intersection, complement, and membership.

4.6. CHARACTER STRING PROCESSING

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Character data type [1]	.	X	X	.	.	.	.	.	.	.	X	.	X	.
Character string data type [2]	.	.	.	X	.	X	.	X	X	.	.	.	X	X
Assignment operator for strings	.	.	.	X	.	X	.	X	X	X	.	.	X	X
Concatenation operator	.	.	.	X	.	X	.	X	.	X	.	.	X	.
Substring pseudo-operator [3]	.	.	.	X	.	X	.	.	X	X	.	.	X	X
Substring function only	.	.	.	.	.	.	.	X	.	.	.	.	.	.
Length function	.	.	.	X	.	X	.	.	.	.	.	.	X	.
Character search function (INDEX)	.	.	.	.	.	X	.	.	.	X	.	.	X	.
Relational operators =, ~=	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Relational operators <, > <=, >=	X	X	X	.	X	X	X	.	.	X	X	X	X	.
Conversion between character and integer data type	X	X	X	X	.	X	.	X	X	X	X	.	X	X

Notes:

[1] Fortran has no character data type, but permits characters to be packed into INTEGER variables. LITTLE is typeless but provides character string operators.

[2] C, CONCURRENT PASCAL, and PASCAL have no character string data type, but they do permit character arrays.

[3] A substring pseudo-operator can appear on the left-hand side of an assignment statement.

4.7. BIT STRING PROCESSING

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Bit data type [1]	.	.	.	.	.	X	.	X	X	X	.	.	.	X
AND, OR, NOT functions	X	X	.	.	.	X	X	X	X	X	.	.	X	X
SHIFT function	X	X	.	.	.	.	X	.	X	.	.	.	X	X
Bit substring pseudo-operator	X	.	.	.	.	X	X	.	X	X	.	.	X	X
Concatenation operator	.	.	.	.	.	X	.	.	.	X	.	.	.	.
Relational operators =, ~=	X	.	.	.	.	X	.	X	.	X	.	.	X	X
Relational operators <, > <=, >=	X	.	.	.	.	.	.	X	.	X	.	.	X	X

Notes:

[1] BLISS-11 is typeless but it provides bit manipulating operators. INTERDATA FORTRAN V and SIMPLT-T have no bit data type but they provide operators or functions for manipulating bits in integer expressions.

4.8. NUMERICAL PROCESSING

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Integer data type	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Floating point data type	.	X	X	X	X	X	X	X	X	.	X	X	.	X
Fixed point data type	.	.	.	X	.	.	.	.	X	.	.	.	.	X
Complex data type	.	.	.	X	X	.	X	.	.	.	.	X	.	.
Double precision floating point type	.	X	.	.	X	X	X	.	X	.	.	X	.	.
Variable precision for all numeric data types	.	.	.	X	.	.	.	.	.	.	.	.	.	X
Automatic conversion between the numeric types	.	X	.	X	X	X	X	.	X	.	.	X	.	X
Generic numerical functions	.	.	X	X	.	X	.	X	X	.	X	.	X	X
Ability to intercept underflow or overflow conditions	.	.	.	X	.	X	.	.	.	.	.	.	.	X
Matrix or vector data type [1]	.	.	.	X	.	X	.	.	.	.	.	.	.	.
Matrix and vector assignment operator	.	.	.	X	.	X	.	.	.	.	.	.	.	X
Matrix and vector relational operators =, ~=	.	.	.	X	.	X	.	.	.	.	.	.	.	X
Matrix and vector dot product	.	.	.	X	.	X	.	.	.	.	.	.	.	X
Vector cross product	.	.	.	X	.	X	.	.	.	.	.	.	.	X
Matrix inverse, transpose, and trace	.	.	.	X	.	X	.	.	.	.	.	.	.	X
FOR or DO loops [2]	X	X	X	X	X	X	X	.	X	X	X	X	.	X

Notes:

[1] SPL has no matrix type, but it provides many operators for manipulating 1 or 2-dimensional sections of arrays.

[2] FLECS, INTERDATA FORTRAN V, LITTLE, and PREST4 require the DO loop increment to be positive. CONCURRENT PASCAL and PASCAL only permit +1 or -1 as the loop increment.

4.9. EFFICIENCY

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Uses runtime stack	X	X	.	X	.	X	.	X	X	.	X	.	X	X
Uses dynamic storage allocator	.	.	.	.	.	.	.	X	.	.	X	.	.	.
Uses system monitor for runtime scheduling	.	.	X	X	.	X	.	.	.	.	.	.	.	.
Structured control structure	X	X	X	X	X	X	.	X	X	X	X	X	X	X
High level data structures and operators	.	.	X	X	.	X	.	X	X	.	X	.	.	X
User requested packing densities [1]	.	.	.	.	.	X	.	.	X	.	X	.	.	X
Bit packing feature in tables or structures [2]	.	.	.	X	.	.	.	.	X	.	.	.	.	X
OVERLAY or EQUIVALENCE stmt	.	.	.	.	X	.	X	.	X	.	.	X	.	X
INLINE attribute for procedures and functions [3]	.	.	.	X	.	.	.	.	X	.	.	.	.	X
Compiler directives for requesting fast storage [4]	X	X	.	.	.	X	.	.	.	.	.	.	.	X
Inline assembly language	X	.	.	X	.	.	X	.	.	.	.	.	.	X

Notes:

[1] Such as the table or record attributes MEDIUM, DENSE, TIGHT.

[2] User allocation of data items within tables or records, including word and bit position.

[3] INLINE attribute to force procedures or functions to be expanded inline instead of generating a calling sequence.

[4] Such as the HAL/S TEMPORARY statement and the C REGISTER statement.

4.10. SPECIAL SYSTEM FEATURES

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Record structure [1]	.	X	X	X	.	X	.	X	X	.	X	.	.	X
Bit manipulating features	X	X	.	X	.	X	X	X	X	X	.	.	X	X
Character manipulating features [2]	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Pointers or indirect addressing [3]	X	X	.	.	.	X	X	X	.	.	X	.	.	X
Access to machine registers	X	.	.	X	.	.	.	.	.	.	.	.	.	X
Inline assembly language	X	.	.	X	.	.	X	.	.	.	.	.	.	X
Reentrant or recursive procedures	X	X	X	X	.	X	.	.	X	.	X	.	X	X
Exception handling constructs [4]	X	.	.	X	.	X	.	.	.	.	.	.	.	X
Special subroutine linkages [5]	X	.	.	X	.	X	.	.	.	.	.	.	.	.
Dynamic storage allocation [6]	.	.	.	.	.	.	.	X	.	.	X	.	.	.
Concurrent processes	.	.	X	X	.	X	.	.	.	.	.	.	.	.
Real-time scheduling of processes	.	.	.	X	.	X	.	.	.	.	.	.	.	.
Ability to access a block of core under varying data formats [7]	X	.	X	X	X	X	X	.	X	X	.	X	.	X

Notes:

[1] JOVIAL only has a table data structure (tables can only be formed from simple items, so that a JOVIAL table can not contain another table or an array as one of its items).

[2] The Fortran languages FLECS, INTERDATA FORTRAN V, and PREST4 provide inadequate character manipulating features.

[3] Pointers in INTERDATA FORTRAN V can only be used to fetch data indirectly, they can not be used to store data indirectly.

[4] Such as ON-conditions or signal handlers.

[5] Subroutine linkages to other languages (like Fortran, PL/I, assembly language), or user control over the subroutine linkage (how arguments are passed, which registers are altered, how result

is returned, and so forth).

[6] Such as the JOSSLE ALLOCATE statement or the PASCAL function NEW.

[7] Without using assembly language routines.

4.11. ERROR CHECKING AND DEBUGGING

Language Feature	Languages													
	BL	C	CP	C4	FL	HS	IF	JS	JV	LI	PA	P4	SI	SP
Complete type checking [1]	.	X	X	X	.	X	.	X	.	.	X	.	X	.
Partial type checking only	.	.	.	.	X	.	X	.	X	.	.	X	.	X
No automatic conversions between the basic data types	X	.	X	.	.	.	.	X	.	X	X	.	X	.
No default type declarations	X	.	X	X	.	.	.	X	.	X	X	.	X	.
Exception handling constructs [2]	X	.	.	X	.	X	.	.	.	.	.	.	.	X
Debugging aids:														
Subscript checking	.	.	.	X	.	.	X	X	.	X	.	.	X	.
Variable tracing	.	.	.	.	.	.	X	.	.	X	.	X	X	X
Calling history	.	.	.	.	.	.	.	.	.	X	.	.	X	X
Execution-time statistics	.	.	.	.	.	.	.	.	.	X	.	.	X	X
Conditional compilation feature	.	.	.	.	.	.	X	.	X	.	.	.	X	.

Notes:

[1] Including procedure parameters and pointer variables.

[2] Such as ON-conditions or signal handlers.

5. RECOMMENDATIONS

5.1. Introduction

Based on our study of POCCNET requirements and our evaluation of the languages, we have concluded that none of the fifteen languages can satisfy all of the requirements. The application areas within POCCNET are diverse and there are too many additional constraints on the implementation language. Since none of the languages satisfy all the requirements, a language (or group of languages) should be chosen that satisfies most of the POCCNET requirements at a low cost.

The requirements of the POCCNET implementation language were discussed in Chapters 2, 3, and 4. They included support for the five application areas, and additional constraints such as machine independence, efficiency, and modifiability. However, we should also consider the costs associated with each of the fifteen languages. Language costs can be subdivided into start-up costs, development and testing costs, and maintenance costs. The start-up costs for the POCCNET language include the cost of obtaining compilers, training personnel in the new language and design methodology, and developing other language tools (such as macro processors, debugging aids, and special linkers or loaders). Start-up cost will therefore be directly affected by the complexity of the language and the availability of compilers for the language.

Development and testing costs will be affected by the design support and debugging features in the language. These include features supporting reliability, modularity, modifiability, readability, and error checking/debugging aids. Type checking of procedure and function parameters will speed the integration testing of program modules.

Maintenance costs will be affected by the readability and modifiability of the language. Languages that are not machine

dependent will require fewer software changes as new hardware is added to POCCNET. Documentation aids such as cross reference and attribute listing, static and execution-time program statistics, and standardized output listings would also lower the cost of maintaining POCCNET software.

Another factor to consider is the relatively long life of POCCNET. The network is expected to support GSFC POCCS throughout the 1980's. Over such a long period the development, testing, and maintenance costs will greatly exceed the start-up costs associated with the implementation language.

5.2. Language Recommendations

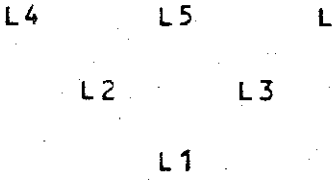
At this point we will discuss our conclusions and recommendations about the fifteen languages. The languages fall naturally into five groups: (1) the SIMPL and PASCAL families; (2) the high level languages CS-4, HAL/S, JOSSLE, JOVIAL, and SPL; (3) the Fortran languages FLECS, INTERDATA FORTRAN V, and PREST4; (4) the low to medium level languages BLISS, C, and LITTLE; (5) the macro assembly language STRCMACS. We will discuss each of these groups in turn.

5.3. Families of Languages

As discussed previously, there are a number of application areas within POCCNET. These range from real-time and general systems programming up to numerical and data base processing. POCCNET poses additional constraints on the implementation language, including machine independence, reliability, and modifiability. Based on our evaluation of the languages, none of them meet all the POCCNET requirements. Moreover, it is likely that any language that did satisfy all of the requirements would be too large and contain too many contradictory features [BAS76b]. The runtime environment needed to support such a language would be complex and inefficient. What we would like

instead is a set of languages, each tailored to one particular subapplication. However, there are several drawbacks to building a large set of independent languages. For one thing, the design and development of new programming languages would be fraught with many problems since each language would be an entirely new design experience. Secondly, if these languages were truly different in design, it would require the user to learn several totally different notations for solving the different aspects of the problem. Thirdly, there would be a proliferation of languages and compilers to maintain.

One possible approach that minimizes some of the above drawbacks is the development of a family of programming languages and compilers. The basic idea behind the family is that all the languages in the family contain a core design which consists of a minimal set of common language features and a simple common runtime environment. This core design defines the base language for which all the other languages in the family are extensions. This also guarantees a basic common design for the compilers. The basic family concept can be viewed as a tree structure in which each of the languages in a subtree is an extension of the language at the root of the subtree. For example:



In this case the language L4 = L2 U {new features of L4}.

Using the family approach permits the development of several application area languages, minimizing the difference between the languages and the compiler design effort. Since many of the constructs for various applications contain a similarity of design or interact with the environment in similar ways, experience derived from one design and development effort can be directly applied to another. Since the best choice of notation for a particular application area may not be known a priori, the family idea permits some experimentation without the cost of a

totally new language and compiler development.

There are several approaches to minimizing the compiler development for a family of languages. One can develop an extensible language and build the family out of the extensible base language. The extension can be made either by a data abstraction facility as in CLU [LIS74] or by some form of full language extension as in ELF [CHE68]. The family of compilers can also be built using a translator writing system or by extending some base core compiler, as was done with the SIMPL and the PASCAL families. A combination of two of the above techniques is recommended here, and they will be discussed a little more fully.

In the core extensible compiler approach, the base compiler for the base language is extended for each new language in the family, creating a family of compilers. In order to achieve the resulting family of compilers, the core compiler must be easy to modify and easy to extend with new features. One experience with this technique, the SIMPL family of languages and compilers, has proved reasonably successful with respect to extensibility due to the use of specialized software development techniques during compiler development.

Using the core extensible compiler approach, the compiler C(L) for a new language in a subtree is built from the compiler for the language at the root of the subtree. This is done by making modifications (mod) to that compiler to permit it to handle the new features of the extension language. For the family in the previous example we have

$$C(L4) = C(L2) \text{ mod } \{L4 \text{ fixes}\} \cup \{\text{new } L4 \text{ routines}\}$$

where the set of L4 routines represents the code for the L4 extensions to L2, and the set of L4 fixes represents the code for modifying the L2 compiler to add those extensions. The key to good extensible compiler design is to minimize the number of modifications (fixes) and maximize the number of independent routines.

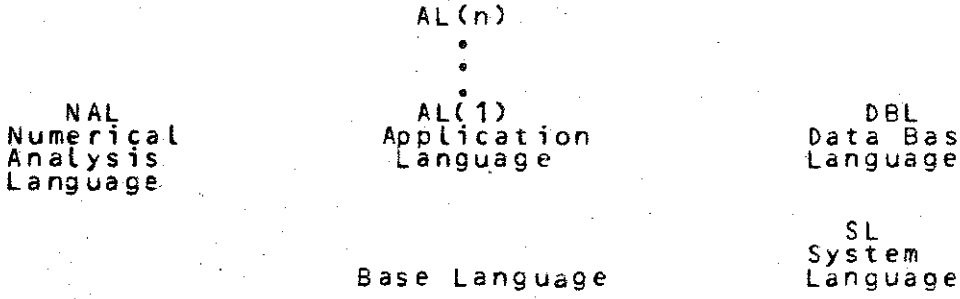
Using a data extension approach, new data types and data structures can be added to the language using a built-in data

abstraction facility. In order to achieve reasonable extensibility, the facility should be easy to use and permit efficient implementation. Experience with forms of data abstraction facilities in CS-4 and CONCURRENT PASCAL have demonstrated the benefits of this approach.

Here the effective compiler for a new language is again built from the compiler for its immediate ancestor in the tree. This is done by adding a new set of library modules that represent the new data types and structures and their associated operators and access mechanisms, respectively. For example, C(L4) = C(L2) U {L4 library modules}.

Each of the two techniques has different assets. The core extensible compiler approach permits full language extension, including new control structures and modifications to the runtime environment. It offers the most efficiency and permits a full set of specialized error diagnostics to be built in. The data definitional approach can be used only for data extensions, but these are by far the most common in the range of subapplication. It is also a lot easier to do and can be performed by the average programmer, where the compiler extensions require more specialized training. Ideally, the first approach should be used for application extensions and the second for smaller subapplication extensions.

Let us now apply this family concept to the POCCNET system and consider how the various application-oriented language features could be distributed across several languages in the family. There would be a language in the family for each application, i.e., a systems programming language, a numerical analysis language, a data base language, a graphics or display language, and so forth. Each language would be built out of some base language (which may in fact be the system language). The application language may have several extensions, each of which adds on some higher level set of primitives. For example, some set of standardized algorithms could be defined as a set of primitive operations in the language. The family tree for the language may take on a form such as



In general, the application languages can be just as high an extension of AL(1) as is appropriate for the sophistication of the user. The system is then modularized so that each module is programmed in the appropriate language, e.g., a numerical analysis module in the numerical analysis language NAL. Each of these modules can interface with the others through an interfacing system. The interface system is part of the basis for the family of languages and contains among other things the compilers for the languages. The interface system could be built into the IPC or Interface Subsystems of POCCNET.

It is clear that the family of languages concept permits the incorporation of the various capabilities required for the POCCNET system. This concept also rates well with respect to design support, reliability, efficiency, machine independence, and reusability.

With respect to ease of expression, algorithms are written in a notation which is specialized to the application. Since each language is reasonably independent of the application level, primitives in one notation can be fine-tuned without affecting the primitives of another application. This permits a certain amount of experimentation, and primitives can be varied with experience.

High level, application-oriented primitives make a solution algorithm easier to read and understand and therefore easier to verify as correct. The specialized notation raises the level of the executing algorithm to the level at which the solution is developed. Debugging features will be improved, because the compilers for the individual languages can tailor error diagnostics and recovery to the particular application.

Each language is small and relatively simple, so that compilation of programs is very efficient. Each language is not complicated by a mix of features whose interaction may complicate the runtime environment, and a simpler runtime environment implies more efficient execution. Language features are specialized to meet one specific application and don't have to be generalized, inefficient versions of the feature. If necessary, the programmer can always use one of the lower level languages to improve or fine-tune an algorithm.

Higher level primitives will make programs more portable. The hierarchy with respect to the data abstractions permits the localization of the machine dependent aspects of the program, and these localized sections can be recoded when the program is transported to a different machine. And with regard to the development of reusable software, each application area has its own language. Thus, needed submodules are written in the target application notation rather than the host application notation. This makes it easier to recognize the essential function of the submodule and easier to write it in a more generally applicable way.

Our primary recommendation is that POCCNET be implemented using a family of languages. Two such families (PASCAL and SIMPL) were examined in this study. However, since neither of these families as they currently exist will satisfy all of the requirements of POCCNET, we recommend that one of the families be improved for POCCNET. The compilers for both languages are written in a high level language (PASCAL and SIMPL-T) and both were designed to be modifiable and machine independent.

The two major deficiencies in the PASCAL family are the lack of external procedures (programs must be compiled en masse) and the lack of "adjustable" arrays or strings as formal procedure parameters. PASCAL requires the length of formal array or string parameters to be declared at compile time, so there is no way to write a PASCAL procedure that will manipulate arrays or strings of arbitrary length. We would recommend that external procedures be added to PASCAL, and that adjustable arrays and strings be

provided using the "*"-bound of PL/I or by passing in the array or string dimensions, as is done in Fortran. At least one implementation of PASCAL on the IBM 360 series already provides external procedures and functions [RUS76]. The usefulness of CONCURRENT PASCAL for systems programming would also be increased by the addition of a bit string data type.

The SIMPL family could be improved by the addition of record structures and multidimensional arrays (both of these extensions have already been designed). The system features in SIMPL-XI could also be extended for POCCNET. The addition of a data abstraction facility would greatly improve the entire SIMPL family. Finally, these changes would not require complete reworking of the compiler, since the SIMPL compiler was specifically designed to be extendible.

5.4. Use of a Single Language

The second alternative for a POCCNET implementation language is to use a single language that meets most of the POCCNET requirements. Any of the languages CS-4, HAL/S, JOSSLE, JOVIAL/J3B, and SPL/Mark IV could be used to implement most of POCCNET.

We recommend that HAL/S be chosen over the other four languages. HAL/S has few machine dependent features, it is efficient, and it has many system features (including records, pointers, real-time process scheduling, and exception handling statements). The language also has features that would improve the reliability and modifiability of programs, including full type checking, COMPOOL files, a macro processor, and structured control structures. HAL/S has been implemented on the IBM 360 series, the Data General Nova, and the Shuttle flight computer.

Although the CS-4 language has many nice features (such as data abstractions), the language is currently under development and no compiler is available. For this reason, we can not recommend CS-4 for use in the POCCNET system. SPL is judged to be equivalent to HAL/S in power, but the language has many

features that would decrease the reliability of programs. SPL provides many low level and machine dependent features, automatic type conversion between all of the basic data types, default declarations of variables, and "implicit" subscripts for arrays. SPL was therefore judged to be inferior to HAL/S. Finally, the JOSSLE and JOVIAL/J3B languages are proper subsets of HAL/S and were therefore eliminated.

5.5. Use of Fortran

Because of its widespread use in the computer industry, this report must discuss the possibility of using Fortran as the POCCNET implementation language. Fortran variants have been implemented on almost all commercial computer systems. Although there are many minor differences between the implementations, almost all implementations support the 1966 ANSI Standard Fortran. In addition, the Fortran language is more widely known than any of the other languages in this study. Thus, the use of Fortran as the POCCNET implementation language would probably permit a shorter start-up time and a lower initial cost than the other languages.

Despite the lower initial cost, we recommend that Fortran not be used for POCCNET. Over the course of a long project like POCCNET, it is likely that the cost of software development and maintenance will greatly exceed the initial start-up cost. This is crucial, because Fortran provides few features that support the development or maintenance of programs. The language has few control structures, so that GOTO and IF statements must be used to simulate if-then-else statements, while loops, case statements, etc. No bit or character data type is provided. Bit and character data must therefore be stored in INTEGER variables, and it becomes impossible to enforce type checking between the integer, bit, and character data types. Fortran also doesn't perform type checking for subroutine or function parameters, so integration testing of Fortran programs becomes more difficult. The only data structure provided by Fortran is the array: the

language has no record structure for forming logical groupings of data. Finally, Fortran has no macro facilities, no CONSTANT statement for defining program parameters, and no INCLUDE feature for copying source files into a program. The lack of these features will make Fortran programs longer than necessary and difficult to read, modify, or debug.

Finally, Fortran does not provide the system features that are required of the POCCNET implementation language. Fortran has no pointers, records, reentrant procedures, access to machine registers, concurrent processes, or exception handling features. Some of these features can be simulated by calling assembly language routines, but with considerable loss in efficiency. For all these reasons, we feel that Fortran would be a poor choice for the POCCNET implementation language.

If Fortran is chosen as the implementation language (in spite of our recommendations), we strongly advise that a preprocessor be used to provide control structures for structured programming. Two such preprocessors (FLECS and PREST4) were examined in this study. Since PREST4 forbids the use of some Fortran constructs (FLECS does not) and provides fewer new control structures than FLECS, we recommend that FLECS be chosen as the Fortran preprocessor. FLECS is written in Fortran and is available from its author, T. Beyer, at a nominal cost ($100). Many other Fortran preprocessors are also available [MEI75].

5.6. Remaining Languages

The remaining languages were eliminated early in the study when it became clear that they did not come close to satisfying all the requirements of the POCCNET system. The languages BLISS-11 and LITTLE were considered to be too low-level for general use in POCCNET. Both of these languages are typeless, systems implementation languages. While the language C provided many low-level features within a typed, medium level language, it was rejected because of its terse and frequently unreadable syntax.

Finally, some portions of the POCCNET system may be written in assembly language where time or space efficiency is critical. For these portions, we recommend that the vendor's assembly language be augmented by a set of structured macros similar to STRCMACS. Macros of this type can greatly improve the readability of assembly language programs. The structured macros can be expanded during the normal assembly step if the assembly language provides a macro facility, or during a preprocessor pass if no such facility exists.

5.7. Summary

To summarize, on the basis of our study none of the fifteen languages meet all of the requirements for a POCCNET implementation language. Our primary recommendation is that a family of languages be developed for POCCNET by modifying the PASCAL or SIMPL families. If a single implementation language is to be used then we recommend that the NASA Shuttle language HAL/S be chosen. We recommend that Fortran not be used as the implementation language. Finally, if Fortran or assembly language are used in POCCNET then preprocessors should be used to provide structured control structures.

th-th-that's all folks!