SLAC-PUB-3207
September 1983
(N)

AN INTRODUCTION TO MODULA-2
(Installation Code: SLA)

Bebo White*

Stanford Linear Accelerator Center
Stanford University, Stanford, California 94305

ABSTRACT

Modula-2 is a general, efficiently implementable systems programming language. While most of its structures are derived from Pascal, it overcomes many of the limitations of that language. It also provides a powerful alternative to Ada. This tutorial will present Modula-2's history and an overview of its features.

Presented to

SHARE 61

Sponsored by the
PASCAL PROJECT
Session A703
New York, New York
August 21-26, 1983

*Work supported by the Department of Energy, contract DE-AC03-76SF00515.

Modula-2 has been variously described as Pascal-2, and 'Pascal for Grown-ups.' While these descriptions border on accuracy, they typically are neither complimentary to Pascal nor do they acknowledge Modula-2's unique features which should allow it to be recognized for its own worth. Modula-2 does address many of Pascal's shortcomings, but in doing so is not a condemnation of Pascal, but more of an example of the evolution of a programming language concept.

This paper will provide an overview of Modula-2 with emphasis on three areas:

- The History and Background of Modula-2
- The Features of Modula-2
- Why Would IBM Users Be Interested in Modula-2?

THE HISTORY AND BACKGROUND OF MODULA-2

Modula-2 ,like Pascal, was developed at the ETH-Zurich under the direction of Niklaus Wirth (Institut fur Informatik). Its development grew largely from a practical need for a general purpose, efficiently implementable systems programming language. The first production use of Modula-2 occurred in 1981. Dr. Wirth's book, Programming in Modula-2, was published by Springer-Verlag in 1982.

Figure 1 shows a "genealogical" chart for some of the modern algorithmic programming languages. The branch that includes Modula-2 shows its roots in Mesa and Modula (which partially answers the question - "Whatever happened to Modula-1?").

The high-level language Modula was the first of Wirth's attempts to break one of the last holds of assembly level programming, namely machine-dependent system programming such as device drivers. It has facilities for multiprogramming and was designed specifically for the PDP-11 computers. Modula provides a limited visibility of the underlying hardware. It introduced the concept of the module (similar to the Ada package) and has the concepts of processes (concurrently executable units which can be explicitly initiated), interface modules (which correspond to monitors and are code sections executed in mutual exclusion), and signals (similar to the queues in Concurrent Pascal).

Many of the concepts in Modula were enhanced by Wirth's experience with Mesa while on sabbatical at XEROX Palo Alto Research Center (PARC). Mesa is one component of a programming system developed at XEROX and is aimed at developing and maintaining a wide range of system and application programs.

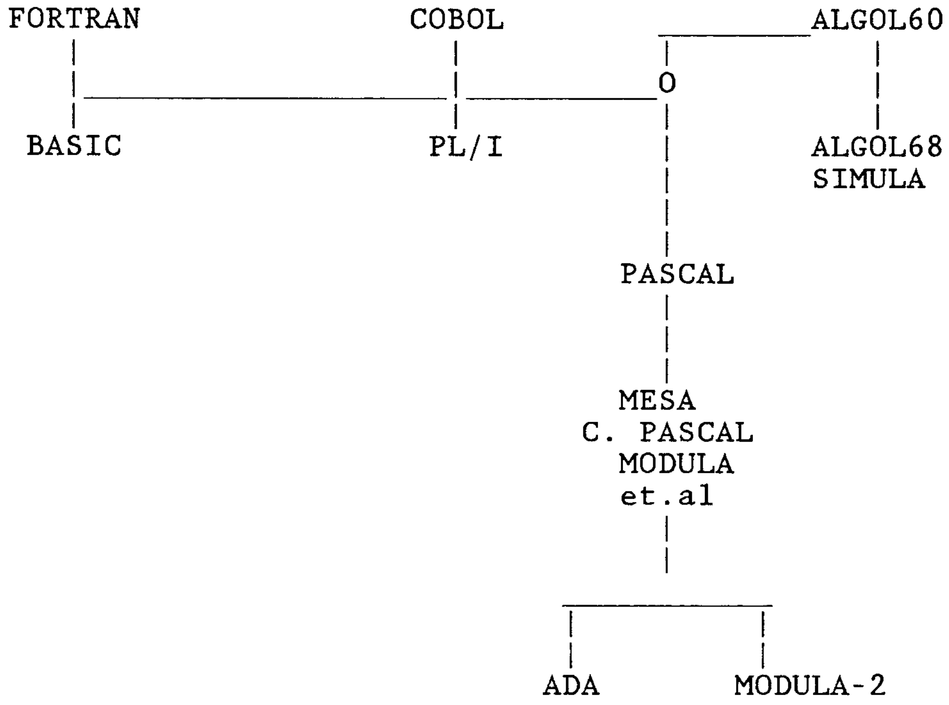

Figure 1: An Evolution of Algorithmic Programming Languages

Modula-2 is the result of experience gained by Wirth from designing, implementing and using Modula. The concept of processes were replaced by the lower level notion of coroutines. The latter permit the programmer to write any desired scheduling algorithm and not be forced, as with Modula, to use the one built into the language for the scheduling of processes. Modula-2 also supports the notion of "programming-in-the-large" by providing separate definition and implementation modules.

It is also important to note in Figure 1 the concurrent development of Modula-2 and Ada. The DoD language survey, which in part prompted the Ada effort, included Modula, as well as Mesa and Pascal. When Wirth implemented Modula-2, he borrowed from Mesa, and was certainly familar with the Ada design work.

However, it was not Wirth's intention to create a language which was a contender with Ada. His goal was to design a computer system (hardware and software) which was capable of being programmed in a single high-level language. This system was given the name Lilith. The programming language used by the Lilith machine had to satisfy requirements of high-level system design as well as those of low-level programming of parts that closely interact with the given hardware. Modula-2 was designed to be that language.

As a result, Modula-2 is essentially machine-independent, with the exception of limitations due to wordsize. This appears to be in contradiction to the notion of a system-programming language, in which it must be possible to express all operations inherent in the underlying computer. This dilemma is resolved with the aid of the module concept. Machine-dependent items can be introduced in specific modules, and their use can thereby effectively be confined and isolated.[1]

THE FEATURES OF MODULA-2

In terms of general features, Modula-2 most closely demonstrates the influence of Pascal. It has adopted most of the data-type concepts of Pascal with some significant additions. Minor variations have been introduced with respect to the Pascal control structures.

Like Ada, Modula-2 is based on four general software engineering concepts:

1. Modularity - all effects are kept as local as possible

2. Data Abstraction - data manipulation is separated from the details of the data structure representation

3. Portability

4. Concurrency control - independent lines of control (processes) are created and synchronized

However, unlike Ada, these features are not obtained via a "more is better" approach. While an Ada compiler may require upwards of 500K bytes, and the Ada manual is in excess of 200 pages, a Modula-2 compiler is running in a 64K machine and is fully described in a 46 page manual. So, it does provide a viable alternative to Ada.

[1] Niklaus Wirth, MODULA-2, ETH Institut fur Informatik Report No. 36, 1980, page 2

Appendix 1 presents a detailed comparison of specific features in Modula-2 with those of Pascal, FORTRAN 77, and PL/I. These languages were chosen because of their availablility on IBM systems and widespread application.

It is perhaps more important to concentrate on those features of Modula-2 which make it unique. It is easiest to demonstrate these features with respect to Pascal, thereby illustrating why Modula-2 is not just an 'Extended Pascal.'

The Role of Modules in Modula-2

Modules are the most important feature distinguishing Modula-2 from Pascal. Relying heavily upon the concepts of "scope" and "block," modules address the problem (usually found in large programs) of separating "visibility" from "existence."

In block-structured languages the range (i.e. program sections) in which an object (e.g. a variable or procedure) is known is called that object's scope, and therefore, defines its visibility. Unfortunately, an object's visibility also binds its existence -- objects created when the block in which they reside is entered are destroyed when the block is exited. It should be possible as an alternative to declare variables that maintain their values, but are visible only in a few parts of a program. Concurrently, there is also a need for closer control of visibility; a procedure should not be able to access every object declared outside of it when it only needs to access a few of them.

Syntactically, modules closely resemble procedures, but they have different rules about visibility and the existence of their locally declared objects. Consider the declarations in the example given in Figure 2.

The only syntactic differences between the module Mod and a normal Pascal procedure declaration are the reserved word beginning the declaration (MODULE instead of PROCEDURE) and the presence of IMPORT and EXPORT declarations following the module heading.

The semantic differences are more interesting. The objects declared within Mod -- a,b,and c -- exist at the same time as the variables x, y, and z, and remain so as long as Outside is active. The objects named in Mod's IMPORT list are the only externally declared objects visible within Mod -- x, but neither y nor z. The objects declared in Mod's EXPORT list are the only locally declared objects visible outside Mod. Thus, a and P1 are accessible throughout Outside, but b and c remain hidden inside Mod.

```
     MODULA-2                          PASCAL
     ________                          ______

 PROCEDURE Outside;                PROCEDURE Outside;
   VAR x,y,z: INTEGER;               VAR x,y,z: INTEGER;

   MODULE Mod;
     IMPORT x;                           (* no module here *)
     EXPORT a,Pl;
     VAR a,b,c: INTEGER;                   a,b,c: INTEGER;

     PROCEDURE Pl;                   PROCEDURE Pl;
     BEGIN                           BEGIN
       a := a + 1;                     a := a + 1;
       x := a;                         x := a;
     END Pl;                         END; (* Pl *)

   END Mod;

   . . .                             . . .

 END Outside;                      END; (* Outside *)
```

Figure 2: Example of Module Declaration

Figuratively speaking, a module can be considered a syntactically opaque wall protecting its enclosed objects, be they variables or procedures. The export list names identifiers defined inside the module that are also to be visible outside. The import list names those identifiers defined outside the module that are visible inside. Generally, the rules for modules are:

1. Locally declared objects exist as long as the enclosing procedure remains activated;

2. Locally declared objects are visible inside the module and, if they appear in the module's export list, they are also visible outside;

3. Objects declared outside of the module are visible inside only if they appear in the module's import list;

The example given in Figure 3 demonstrates the essence of modularity.

```
    MODULA-2                          PASCAL
    ________                          ______

MODULE MainProgram;               PROGRAM MainProgram;
                                    VAR Seed : INTEGER;
     . . .

  MODULE RandomNumber;                . . .
    IMPORT TimeOfDay;
    EXPORT Random;
    CONST Modulus = 2345;
          Increment = 7227;
    VAR Seed : INTEGER;             FUNCTION Random : INTEGER;
                                      CONST Modulus = 2345;
    PROCEDURE Random() : INTEGER;           Increment = 7227;
    BEGIN                           BEGIN
     Seed := (Seed+Increment)         Seed := (Seed+Increment)
             MOD Modulus;                      MOD Modulus;
     RETURN Seed;                     Random := Seed;
    END Random;                     END; (* Random *)

  BEGIN (* RandomNumber *)
    Seed := TimeOfDay;                 . . .
  END RandomNumber;
      . . .

BEGIN (* MainProgram *)           BEGIN (* MainProgram *)
                                    Seed := TimeOfDay;
      . . .                         . . .
  WriteInt(Random(), 7);            Writeln(Random, 7);
      . . .                         . . .
END MainProgram.                  END. (* MainProgram *)
```

Figure 3: The Essence of Modularity

The random number generator in these examples uses its previous value as a seed variable to generate the next random number. Thus, that value must be maintained across function calls. The program on the right shows the classical Pascal solution. Notice that Seed's declaration is at the top of the program, while its initialization is forced to the bottom. Two obvious disadvantages arise from the scattering of Seed across the face of the program:

1. Its occurrences become hard to find, especially in a large program;

2. It becomes accessible to every other procedure in the program, even though it is used only by Random;

The example on the left demonstrates the usefulness of the module structure. The only object visible to the outside world is the procedure Random, while all objects pertaining to the random number generator are contained in one place. Note that the module RandomNumber contains both declarations and a statement part. Module bodies are the (optional) outermost statement parts of module declarations and serve to initialize a module's variables. Although subject to the module's restrictive visibility rules, module bodies conceptually belong to the enclosing procedure rather than to the modules themselves. Therefore, module bodies are automatically executed when the enclosing procedure is called.

Relaxed Declaration Order

New Pascal users are often frustrated and confused by the enforced declaration and definition block structure required within the program skeleton. Despite the emphasis on modules, blocks still play an important part in Modula-2: implementation modules, program modules, internal modules, and procedures are all declared as blocks. Differences from Pascal include relaxed order of declarations, termination of all blocks by a procedure or module identifier, and the optional nature of block bodies.

Pascal imposes a strict order on the declaration of objects; within any given block, labels must be declared before constants, constants before types, and so on. Modula-2 eliminates this restriction -- declarations can appear in any order. Programs containing a large number of declarations are easier to read and understand when related declarations are grouped together (regardless of their kind).

The following is an example of relaxed declaration order:

```
MODULE Xlator;
  CONST MaxsSym = 1024;
  TYPE SymBuffer = ARRAY[1..MaxSym] OF CHAR;
  VAR SymBuff1, SymBuff2: SymBuffer;
  . . .
  CONST MaxCode = 512;
  TYPE CodeBuffer = ARRAY[1..MaxCode] OF BYTE;
  VAR CodeBuff: CodeBuffer;
  . . .
END Xlator.
```

This example easily demonstrates how various related declarations may be placed together in a Modula-2 program, whereas in a Pascal program they may be scattered due to strict block ordering. Relaxed declaration order not only improves readability but also enables a logical ordering which may be very important in large programs.

Separate Compilation

Separate compilation is allowed by the Modula-2 compiler through the use of the "compilation unit." Modula-2 programs are constructed from two kinds of compilation units: program modules and library modules. Program modules are single compilation units and their compiled forms constitute executable programs. They are analogous to standard Pascal programs.

Library modules are a different animal and form the basis for the Modula-2 library. They are divided into a definition module and an implementation module. Definition modules contain declarations of the objects which are exported to other compilation units. Implementation modules contain the code implementing the library module. Both always exist as a pair and are related by being declared with the same module identifier.

To understand the rationale behind dividing a library module into separate definition and implementation modules, consider the design and development of a large software system, such as an operating system. The first step in designing such a system is to identify major subsystems and design interfaces through which the subsystems communicate. Once this is done, actual development of the subsystems can proceed, with each programmer responsible for developing one (or more) of the subsystems.

The specification of a (program) module may be viewed as a contract between the user of the module and its implementer. It must contain all the information needed to:

1. Enable the user to design a program that uses the module, and verify its correctness, without knowing anything about how the module is implemented.

2. Enable the implementer to design a module, and verify its correctness, without knowing anything about the program that uses the module.[2]

Now consider the project requirements in terms of Modula-2's separate compilation facilities. Subsystems will most likely be composed of one or more compilation units. Defining and maintaining consistent interfaces is of critical importance in ensuring

[2] Leslie Lamport, 'Specifying Concurrent Program Modules,' ACM Transactions on Programming Languages and Systems, Vol. 5, No. 2, April 1983, pages 190-222

error-free communication between subsystems. During the design stage, however, the subsystems themselves do not yet exist. They are known only by their interfaces.

The concept of a subsystem interface corresponds to the definition module construct. Thus, interfaces can be defined as a set of definition modules before subsystem development (i.e., design and coding of the implementation modules) begins. These modules are distributed to all members of the programming group,and it is through these modules that subsystem interfaces are defined. Interface consistency is automatically enforced by the compiler.

Modula-2 Libraries

The library is a collection of separately compiled modules that forms an essential part of most Modula-2 implementations. It typically contains the following kinds of modules:

1. Low-level system modules which provide access to local system resources;

2. Standard utility modules which provide a consistent system environment across all Modula-2 implementations;

3. General-purpose modules which provide useful operations to many programs; and,

4. Special-purpose modules which form part of a single program.

The library is stored on one or more disk files containing compiled forms of the library module's compilation units. The library is accessed by both the compiler and the program loader - the former reads any required (pre-compiled) definition modules during compilation, then the latter loads the corresponding implementation modules during execution.

A dependency exists between library modules and the modules that import them. Consider the example of a single library module. The compiler must reference the module's symbol file (a compiled definition module) in order to compile the implementation module. Therefore, the definition module must be compiled first. Once an implementation module has been compiled, its object file is tied to the current symbol file, since the object code is based on procedure and data offsets obtained from the symbol file. Similarly, when a program imports a library module, it is assumed that the symbol file offsets are accurate reflections of the corresponding object file.

The Modula-2 language contains no pre-defined (standard) procedures for I/O, memory allocation, or process scheduling. Instead, these facilities are provided by standard utility modules stored in the library. The contents of a standard library would be expected to include:

- Storage management
- Format conversions (such as binary to text and vice-versa)
- Console I/O (keyboard polling)
- Directory/file operations (reading and writing byte streams of arbitrary types, random-access, etc.)
- Code management
- Mathematical functions
- Strings and related manipulation functions
- Etc., etc., etc. (Wirth has stated that library modules are "...an essential part of a Modula-2 implementation."

Standard utility modules are expected to be available in every Modula-2 implementation. Thus, by using only standard modules, Modula-2 programs become portable across all implementations.

The advantages of expressing commonly-used routines as library modules (rather than part of the language) include a smaller compiler, smaller run-time system, and the ability to define alternative facilities when the standard facilities prove insufficient. Disadvantages include the need to explicitly import and bind library modules, and the less flexible syntax required for coding standard operations as library modules (as opposed to their being handled by the compiler).

WHY WOULD IBM USERS BE INTERESTED IN MODULA-2?

It appears that Modula-2 should be of interest to IBM users because of its potential for being the first operating-system-independent high- level language. This would insure portability across the range of IBM systems as well as compatible interfaces with non-IBM peripherals. Such interfaces could be accomplished by placing all machine-dependent features within module libraries.

At the Stanford Linear Accelerator (SLAC), one particular interest in Modula-2 is for an application to support networking (such as Ethernet) on the 3081 under VM/SP. The software protocols and interface to Ethernet could all exist in a single VM running under CMS, with access to the CMS file system and access to other VMs via VMCF (the Virtual Machine Communications Facility, which is part of the CP component of VM/SP; it provides virtual machines with the ability to send data to and receive data from any other virtual machine), via IUCV (the Inter-User Communications Vehicle, which is a communications facility that allows users to pass any amount of information; IUCV enables a program running in a virtual machine to communicate with other virtual machines, with a CP system service, and with itself), and via the virtual reader and punch.

The software would have to be able to listen to the Ethernet channel interface, to VMCF messages, and to reader tag records. The most obvious way to provide such listening is via multitasking with the ability to wait until some other task sends a signal, and the ability to respond to interrupts and execute processes in parallel. Similar software would need to run in other computers, e.g. VAXs or IBM PCs or SUN or STAR workstations) or workstations connected to Ethernet that wish to communicate with the 3081 for file transfers, etc. Thus, the code needs to be portable.

The need for portability means that only a small fraction of the code should be written in assembler language. It is also unrealistic to expect to find experts on network programming who also know assembler language for all the possible machines that will be supported by the network. Current high level languages that exist on both the IBM 3081 and on machines such as a VAX are limited to FORTRAN, Pascal, and C. The FORTRAN and Pascal implementations have no multitasking capabilities. C implementations for VM/SP are just being delivered, and it is not known what interfaces to the system they have to support multitasking.

In the meantime the interface is being coded in Pascal (it was chosen over FORTRAN due to its superior data structures) and targeting VM/SP, the VAX/VMS and the IBM PC. The need for portability requires that the use of assembler code be kept to a minimum.

Appendix A

COMPARISON OF PASCAL, FORTRAN 77, PL/I, AND MODULA-2

	PASCAL	FORTRAN 77	PL/I	MODULA-2
Constants	Integer, real, character, Boolean, character string No expressions (PASCAL/VS allows representation in hex)	Integer, real, double precision, complex, logical, character string Expressions are allowed in the PARAMETER statement	Integer (16 & 32 bits), real (16 & 32 bits), character (8 bit byte), Boolean (bit string), decimal (1-16 digits), label	Integer, real, character, Boolean, character string Expressions are allowed
Types	Possible to define them Well checked	Very limited No way of defining new ones Badly checked	No way of defining new ones	Possible to define them Well-checked
Simple types	Integer, real, character, Boolean, subrange	Integer, real, double precision, logical, complex, character	Same as constants	Integer, real, character, Boolean, cardinal, bitset, subrange
Enumerated Types	Do exist but only with identifiers	No	No	Exist with identifiers and characters Allow the definition of character codes
Type conversions	Only integer-real-character and integer-scalar	Functions exist for most cases	Automatic built-in functions, dummy arguments	Use the type name as a conversion function
Variable initialization	No standard means (VALUE in PASCAL/VS)	Possible with DATA (VSFORTRAN allows in type specification statement)	Yes, in DECLARE statement with INIT (value)	No
Arrays	Subscripts: subrange of integers, characters, scalars, and Booleans Any components	Integral positive and negative subscripts Simple components Maximum 7 subscripts	Positive, negative, integral constants, variables, expressions (subscripts)	Subscripts as in Pascal Any components
Conformant arrays (dynamic arrays)	Defined in Pascal ISO (rare implementation)	Variable dimensions Not easy to use for multidimensional arrays	Yes, use * in subprogram for bound	Yes
Records	Hierarchical definition without restriction	No, but may be implemented by using the internal file input/output mechanism	Hierarchical definition without restriction (called STRUCTUREs)	Hierarchical definition without restriction

	PASCAL	FORTRAN 77	PL/I	MODULA-2
Record tag variants	Yes, but not checked	No, but may be implemented by using the internal file input/output mechanism	Yes, a STRUCTURE with DEFINED attribute	Yes, checked
Sets	Yes, but restrictions due to the implementation: set of characters, integer, scalar limited	No	No, but can be simulated with bit strings	As in Pascal
Pointers	Do exist, but not very well-checked	No	Yes, POINTER type - use with ADDR function BASED attribute CONTROLLED attribute	Do exist and are well-defined
Character packing inside a memory	Exists with PACKED	Depends on the implementation Cannot be checked by programmer	Not needed 1 byte = 1 character	Yes
Dynamic allocation	Does exist and runs well	No	Yes, with ALLOCATE function for CONTROLLED variable	Allocator module in library
Dynamic releasing	Exists but not always well implemented	No	Yes, with FREE function	Deallocator module in library
Character strings	Packed array of characters Use is not very versatile (PASCAL/VS has type STRING)	Exist and specific operations are available	Exists as an entity and specific operations are available	STRING type in library
Operators	+ - * / DIV MOD AND OR NOT IN (PASCAL/VS has | & && ¬)	+ - * / ** .AND. .OR. .NOT. //	+ - * / ** NOT AND OR	+ - * / DIV MOD AND OR NOT IN
Control statement syntax	Based on the compound statement BEGIN...END	Keyword at the end of a statement	DO...END groups BEGIN...END blocks	Based on the compound statement Requires explicit END delimiter No BEGIN...END

Selective statements	IF...THEN...ELSE...
	CASE statement
		IF...THEN...ELSE... END IF
		Computed GOTO
			IF...THEN...ELSE
			SELECT
				IF...THEN...ELSIF... THEN...ELSE...END
				CASE statement

	PASCAL	FORTRAN 77	PL/I	MODULA-2

Loops	FOR v:= E1 TO E2 DO...
	DOWNTO
	REPEAT...UNTIL...
	WHILE...DO...
		DO n,m1,m2,m3
		n CONTINUE
			DO I=m1 TO m2 BY m3;
			DO WHILE...END;
				WHILE..DO..END
				REPEAT..UNTIL..
				LOOP..END
				FOR v:= e1 TO e2 [BY e3] DO..END

Loop exits	None except GOTO (PASCAL/VS has LEAVE)	No (except GOTO)	No (except GOTO)	EXIT statement
Loop continue statements	No (Yes in PASCAL/VS)	No	No (except GOTO)	No
Jumps	GOTO (limited use) (PASCAL/VS has RETURN)	GOTO and ASSIGNed GOTO	GOTO (LABEL parameter can be target of a GOTO in calling program from subprogram)	No
Subprograms	Procedures functions	SUBROUTINES, FUNCTIONS Function statements are available Multiple entry points and return points are available	Procedures Functions	Procedures (can be used as functions)
Parameter transmission	By value By reference Formal procedures and functions (PASCAL/VS has CONST)	By reference Formal subprograms (EXTERNAL statement)	By reference only	By value By reference Procedure type IMPORT EXPORT
Recursion	Yes	No	Yes, a subprogram with RECURSIVE option	Yes
Local variables	Yes	Can be dynamic or static (own variables)	Yes, can be dynamic or static	Yes
Static levels (nested procedures)	Depends on implementation but at least 5	No	Yes	Yes
Abstract types	No, but record types and enumerated types approach abstract typing	No	No	Exist in library modules

	PASCAL	FORTRAN 77	PL/I	MODULA-2
Parallelism (tasks)	No	No	No (use ATTACH macro)	Coroutine concept TRANSFER routine IOTRANSFER routine
Exceptions	No	A few ones exist for input/output operations	Many	Exists in library modules
Type parametrization	No	No	No	Exists in library module
Input/output	Standard procedures Formats only in output	Very complete Numerous input and output formats	Stream (GET, PUT) Record (READ, WRITE)	Numerous modules in standard library Input and output formats
Files	Sequential Same type components Text files (PASCAL/VS allows PDS input/output)	Sequential and direct Binary or text files (VSFORTRAN allows PDS input/output)	Stream, Sequential, INDEXED, DIRECT	Standard library Easily modifiable and expandable
Direct access	Not standard (allowed in PASCAL/VS)	Standard	Standard	Exists in library module
Interactive facilities	Not standard Not very easy to use (PASCAL/VS has INTERACTIVE, TERMIN and TERMOUT)	No problem	Easy	Exists in library module
Interface with operating system	Depends on the implementation (Good in PASCAL/VS)	Depends on the implementation	Depends on the implementation	OS exists as library module
Separate compilations	Not standard Available on most computers (allowed in PASCAL/VS)	Possible; no check	Yes; Libraries (PDS) EXTERNAL attribute	Very good with excellent checking DEFINITION module IMPLEMENTATION module
Low level concepts (hardware dependent)	No (Allowed in PASCAL/VS)	Memory transfers and conversions are possible	Bit level operations	Exists in library module

Appendix B

REFERENCES

1. Niklaus Wirth, Programming in Modula-2, Springer-Verlag, 1982

2. Niklaus Wirth, MODULA-2, ETH Institut fur Informatik Report No. 36, 1980 (available from Volition Systems, P.O. Box 1236, Del Mar, CA 92014)

3. Niklaus Wirth, The Personal Computer Lilith, ETH Institut fur Informatik Report No. 40, 1981 (available from DISER, Inc., 385 East 800 South, P.O. Box 70, Orem, UT 84057)

4. Roger Sumner and Rich Gleaves, "Modula-2 -- A Solution to Pascal's Problems," Journal of Pascal and Ada' September/October 1982

5. Joel McCormack and Rich Gleaves, "Modula-2, A Worthy Successor to Pascal," BYTE April 1983

6. Lee Jacobson and Bebo White, "Introduction to Modula-2 For Pascal Programmers," Pascal News July 1983

7. T. De Marco, "Modula-2: Why It Matters", The Yourdon Report Volume 6, No. 2

8. Niklaus Wirth, "Modula-2 Adds Concurrency to Structured Programming", Electronic Design July 23, 1981

9. Nadia Magnenat-Thalmann, "Choosing an Implementation Language for Automatic Translation", Computing Languages Vol. 7, 1982 (this article provided the model for Appendix A; original contains an evaluation of Pascal, FORTRAN 77, C, and Ada)

10. IBM, VS FORTRAN Application Programming: Language Reference GC26-3986-2, pages 225-231 (used in construction of Appendix A for VS FORTRAN features)

11. IBM, Pascal/VS Programmer's Guide SH20-6162-1, pages 127-128 (used in construction of Appendix A for Pascal/VS features)