

132
Implementing Run-Time Support for Modula-2

by

Wai-Sum Christopher Li

B.A. Ottawa University, 1986

A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE


Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

Approved by:


Major Professor

Dr. Virgil Wallentine

LD
2668
.P4
CMSC
1988
L5
C. 2

Table of Contents

Chapter 1 Overview.....	1
1.1 Introduction.....	1
1.2 Scope of the Project.....	2
1.3 Getting Acquainted with the UNIX PC.....	2
1.4 An Anatomical View of the UNIX System Structure.....	3
1.5 Modula-2 Compilation Process.....	6
1.6 The Mechanisms of Modula-2 Compilation.....	6
1.7 Porting the Modula-2 Compiler.....	9
Chapter 2 The Modula-2 Language.....	13
2.1 An Introductory Example.....	13
2.2 A Comparison between Pascal and Modula-2.....	15
2.3 Library Modules.....	16
2.4 The Process Concept of Modula-2.....	17
2.5 Type Coercion.....	18
2.6 The Prospects of Modula-2.....	19
Chapter 3 Run-Time Libraries in C.....	20
3.1 Introduction.....	20
3.2 Adapted Features in the C Library Modules.....	21
Chapter 4 Coroutine Primitives.....	22
4.1 Introduction.....	22
4.2 Architectural Overview.....	22
4.3 UNIX PC Microprocessor.....	23

4.4	VAX Central Processor.....	25
4.5	Functional Specifications.....	29
4.6	Conventional Differences in Implemenations....	34
Chapter 5	The Standardization of Modula-2 Libraries.....	39
5.1	Draft BSI Standard I/O Library of Modula-2.....	39
5.2	Layering.....	40
5.3	Extensibility.....	41
5.4	Robustness.....	41
5.5	The Semantics of the Top Level Interface	42
5.6	Channels.....	42
5.7	Draft BSI Standard Utility Library of Modula-2.....	44
Chapter 6	Modula-2 Libraries on the UNIX PC	48
6.1	Introduction.....	48
6.2	The Implemenation in Assembly.....	49
6.3	The Implementation in C.....	51
6.4	The Implementation in Modula-2.....	52
Chpater 7	Conclusion and Future Work.....	58
7.1	Conclusion.....	58
7.2	Future Work.....	59
	Bibliography.....	62
Appendix 1	Overview of C Library Routines.....	65

Appendix 2	Draft BSI Standard I/O Library of Modula-2.....	73
Appendix 3	BSI Modula-2 Utility Library Draft on 07 December 1984.....	79
Appendix 4	Listing of Programs.....	98

List of Figures

Figure 1	Architecture of a UNIX System.....	4
Figure 2	The Modula-2 Compilation Process.....	6
Figure 3	The Modula-2 Compilation Mechanism.....	7
Figure 4	The Porting Process of the Modula-2 Compiler.....	10
Figure 5	A Listing of Run-Time Library Utilities.....	12
Figure 6	An Introductory Modula-2 Example.....	14
Figure 7	A Comparison Between Pascal & Modula-2	16
Figure 8	Programmer's Model of MC68010.....	24
Figure 9	Coroutine Control Block Created by NEWPROCESS.....	30
Figure 10	The Mechanism of TRANSFER (source, destination).....	33
Figure 11	The Register Set of VAX Central Processor.....	35
Figure 12	Two Addressing Modes of MC68000.....	36
Figure 13	Examples of the VAX Central Processor and MC68010 Syntax.....	38
Figure 14	The Assembly Instruction of Pushing Parameters on the Stack.....	50
Figure 15	The Stack Configuration of MC68010.....	51

Figure 16	A Modula-2 CASE Statement.....	53
Figure 17	An Application of CASE Statement.....	54
Figure 18	Levels of Data Abstraction.....	57

Chapter 1

Overview

1.1 Introduction

The newly developed language by Dr. Niklaus Wirth, Modula-2, arouses many computer scientist's interest. According to a survey done by the Modula-2 Users Association in August 1986 [1], at least thirty-two universities are currently using Modula-2 in their courses ranging from freshmen to graduate level. More than twelve Modula-2 compilers on eighteen different computer architectures or operating systems spread across four countries are being employed in the academic, commercial or research institutions. The statistics by no means cover the global Modula-2 activities, but they reflect the popularity and potential of the language. Indeed, the modularity of the language and the capability of data abstraction establish a milestone in computer programming language development. Our goal is to make Modula-2 accessible on the UNIX PC's for research and academic purposes.

[1] Mazlack, Larry, "Academic Modula-2 Survey", The MODUS Quarterly, issue #6, November 1986, pp. 37 - 41.

1.2 Scope of the Project

This project encompasses implementing the run-time libraries of Modula-2 in C, the DEFINITION MODULES and IMPLEMENTATION MODULES in Modula-2, and the coroutine primitives in Assembly language of MC68010 microprocessor.

Since the standardization of Modula-2 libraries is underway, the proposal for the standard libraries from the British Standard Institute is discussed in contrast to our implementation. The comparison may shed some light on the directions that this project is heading. However, the proposed standard libraries will not be actually implemented in this project because no consensus has been reached on the standards yet. Alterations are still being made on the proposal.

1.3 Getting Acquainted with the UNIX PC

The AT&T UNIX PC model 3B1 has 2 Megabytes of main memory, a 40 Megabyte disk drive, and a StarLAN interface. It also comes with a mouse which makes window and graphics operations more convenient to perform. Furthermore, it supports application programs like office productivity, business management, communications, and system programming.

Being a work station with network connections for resource sharing makes a UNIX PC a very powerful tool. Since it has its own processor, it allows direct access to the languages and programming tools in the UNIX PC's operating system environment, UNIX System V (Refer to section 1.4). At the same time, connecting with other hosts enables UNIX PC's to access remote files, to transfer files, and to send electronic mail between hosts; it can also process data items in a distributed database spread over many hosts, and provide remote printing capability.

1.4 An Anatomical View of the UNIX System Structure

Figure 1 depicts the high-level architecture of the UNIX system. The hardware at the center of the diagram provides the operating system with basic services such as interrupt and exception handling. The operating system interacts directly with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies. Viewing the system as a set of layers, the operating system is commonly called the system kernel, or just the kernel, emphasizing its isolation from user programs. Because programs are independent of the underlying hardware, it is easy to move them between UNIX systems running on different hardware, if the programs do not make assumptions about the underlying hardware. For

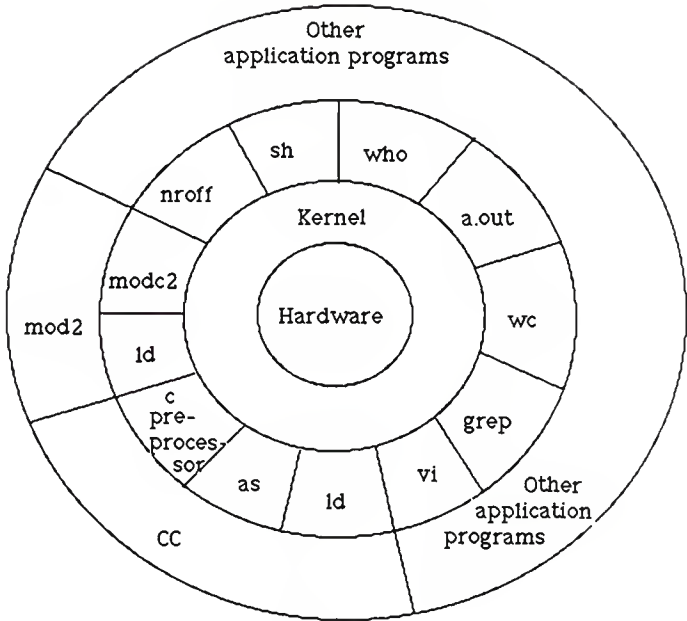


Figure 1. Architecture of a UNIX System

instance, programs that assume the size of a machine word are more difficult to move to other machine than programs that do not make this assumption.

Programs such as the shell and editors (ed and vi) shown in the outer layers interact with the kernel by invoking a well defined set of system calls. The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program. Several programs shown in Figure 1 are in standard system configurations and are known as commands, but private user programs may also exist in this layer as indicated by the program whose name is a.out, the standard name for executable files produced by the C compiler. Other application programs can build on top of lower-level programs, hence the existence of the outermost layer in the figure. For example, the Modula-2 compiler in this project, mod2, is in the outermost layer of the figure; it invokes a Modula-2 one-pass compiler, modc2, and loader, ld, (link-editor), all separate lower-level programs. Although the figure depicts a two-level hierarchy of application programs, users can extend the hierarchy to whatever levels are appropriate. Indeed, the style of programming favored by the UNIX system encourages the combination of existing programs to accomplish a task.

1.5 Modula-2 Compilation Process

The compiler that we are working with is Wirth's compiler [2], a one-pass Modula-2 compiler for MC68000-based computers. The compilation process comprises two main stages : syntax checking in which the native MC68010 machine instructions are generated at the same time--code generation, and linking. (Refer to figure 2.)

Syntax Checking + Code Generation --> Linking
--> Executable Code

Figure 2 The Modula-2 Compilation Process

1.6 The Mechanisms of Modula-2 Compilation

In actuality, the compilation process is not quite as simple as shown in Figure 2. There are more files involved in the process than just the source file and the executable object file. A more detailed diagram is shown in Figure 3.

When a Modula-2 source file is being compiled, it will first be checked for syntax errors. At this point, the compiler will refer to the symbol files when it comes

[2] This Modula-2 compiler was written by Dr. Niklaus Wirth of Institut für Informatik, Swiss Federal Institute of Technology, Zurich, Switzerland.

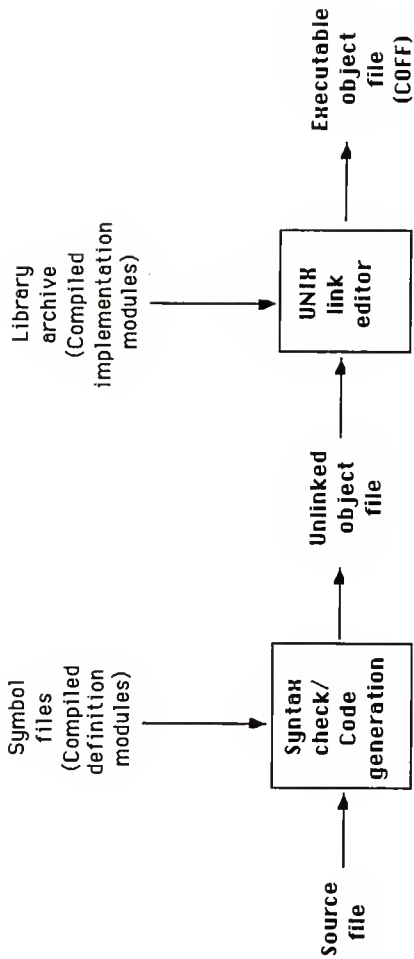


Figure 3 The Modula-2 Compilation Mechanism

across any calls for imported objects which may be procedures, functions, types, constants, or even variables. The symbol files are the compiled DEFINITION MODULES in which all the exported objects are declared [3]. In so doing, any invalid procedure or function calls, misuse of variables, or incompatibility of types can be uncovered when they are compared against the symbols files.

After filtering all the syntactical errors, an unlinked object file will be generated. The reason the object file is unlinked is that all the addresses of the imported elements have not yet been specified. They have only been checked for syntactical validity, but there is still no indication of where to look for the implementation for each imported element. The UNIX link editor will help resolve this dilemma. It is also an issue of relocation, which simply means, in this context, the implementation of the functional objects are separate from the compiler module. Hence, modifications made on the implementation modules will not directly affect the compiler. Compilers written with the relocation technique then become more portable.

All the implementation modules must have been

[3] Procedure declarations consist of a procedure heading only; whereas types, variables, and constants must be declared in full details.

compiled and archived to a library archive file at this point. The UNIX link editor then links the unlinked object file with the library archive file and resolves all the external references which can be found in the archive file. If there are no more error messages, an executable object file (COFF, Common Object File Format) will be generated. This complete object file can then be loaded and executed.

1.7 Porting the Modula-2 Compiler

At present, there is not a Modula-2 compiler available on the UNIX PC. The construction of one for the UNIX PC is now underway. This compiler is written in Modula-2 then compiled and linked by the Modula-2 compiler (m2c) and link editor on the AT&T 3B15. The executable object file generated is the UNIX PC cross-compiler which runs on the AT&T 3B15 to compile the UNIX PC compiler source once again to generate the unlinked object file. This unlinked object file is specially designed for the UNIX PC; it can then be physically ported to a UNIX PC. (See Figure 4 for details of this process.)

Before a UNIX PC Modula-2 compiler can be constructed, the unlinked object file has to have all its undefined external references resolved. In other words,

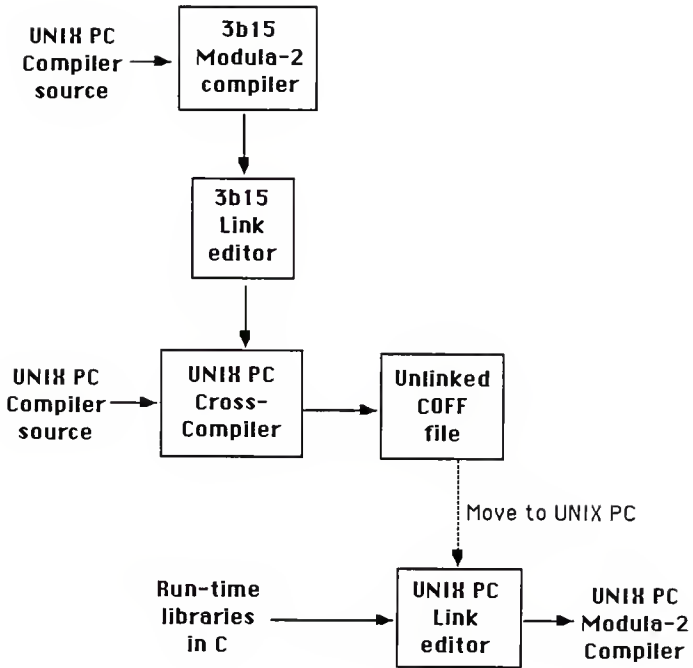


Figure 4 The Porting Process of the Modula-2 Compiler

run-time libraries have to be linked with the unlinked object file by the UNIX PC link editor to supply the external references. In order for the imported elements to execute properly, the run-time libraries must be written in a language available on the UNIX PC; this is necessary before the UNIX PC Modula-2 compiler is generated. For the time being, the only high-level language processor that is available on our UNIX PC's is UNIX System's C compiler [4]. C was chosen to be the language for the run-time libraries. After linking, a UNIX PC Modula-2 compiler results. The whole process is captured in Figure 4.

In order for the Modula-2 compiler to function properly, the run-time libraries in C must include some file system utilities, operations on storage space, interactions with the terminal (eg. standard input and output), and some other miscellaneous utilities. The table below summarizes the procedures, functions, types, variables, and constants that are going to be included in the run-time libraries.

[4] The UNIX System's C compiler, cc, is part of the UNIX System Software Generation System (SGS). It converts C programs into Assembly language programs that are ultimately translated into object files by the assembler, as. The link editor, ld, collects and merges object files into executable load modules.

InOut : WriteString, WriteInt, WriteCard, WriteHex,
WriteLn, WriteOct, Write, EOL.

Storage : ALLOCATE, DEALLOCATE.

DateTime: TimeType, Time.

Terminal: WriteString, Read, WriteLn, Write.

FileSystem
: Create, Delete, Open, Close, Done, ShowStatus,
WriteWord, WriteShortWord, WriteByte, WriteChar,
WriteRecord, ReadChar, ReadWord, SetPos, GetPos,
ModeType, File, CreateTempfile.

Arguments
: ArgCount, GetArgument.

Figure 5 A Listing of Run-Time Library Utilities

Chapter 2

The Modula-2 Language

The Modula-2 language was developed by Professor Niklaus Wirth at the Technical University of Zurich. The language was a result of previous experience with the languages Pascal and Modula (a small experimental language). Being a direct descendent of Pascal, Modula-2 has inherited most of its semantics and some of its syntax from that language. The differences between Modula-2 and Pascal will be discussed in a later section 2.2.

2.1 An Introductory Example

Figure 6 shows a small example of a complete Modula-2 program. All reserved words and standard identifiers are written in uppercase letters while all user-defined names are in mixed case.

```
MODULE myprogram;
  FROM InOut IMPORT WriteString, ReadCard, , WriteReal,
                    ReadReal, Done;

  VAR x,y : REAL;
      i,k : CARDINAL;

  BEGIN
    REPEAT
      WriteString("Enter value for 'x:");
      ReadReal(x);
      IF Done THEN
        WriteString("Enter value for 'i:");
        ReadCard(i);
        IF Done THEN
          y := 1.0;
          FOR k := 1 TO i DO y := y * x END;
          WriteString("(x**i = ");
          WriteReal(y, 8);
        END;
      END;
    UNTIL NOT Done;
  END myprogram.
```

Figure 6 An Introductory Modula-2 Example

The Modula-2 language is highly modular. A module starts with an `IMPORT` list which declares the set of separate modules needed inside the current module. The rest of the code may look quite familiar for Pascal programmers. Nevertheless, there are several noteworthy points. Note that the `IF` statements are closed by a matching `END`, no matter how many statements appear in the `THEN` branch; there is no `BEGIN` to go with every `END`, except the one enclosing the body of the main program. At the end of the program, `END` is followed by the name of the `MODULE`.

2.2 A Comparison between Pascal and Modula-2

Besides the differences mentioned in the previous section, there are some deficiencies of Pascal that are accommodated in by the current features of Modula-2. Figure 7 is sketched to display the contrast between Pascal and Modula-2.

Specific shortcomings of Pascal include :

1. Fixed size arrays.
2. The absence of static variables, other than global variables. The use of global variables often forces a scope larger than desired.
3. No "else" clause in the case statement.
4. No facilities for separate compilation.
5. No facilities for data hiding. Subprograms are bound to a particular data representation. This limits the level of data abstraction that is possible.
6. The declaration order (i.e., constants, types, variables, subprogram) inhibits declarations from being positioned near their point of application.
7. Type checking can never be suppressed. This facility might be desired in exceptional circumstances.
8. No facilities for concurrent processing.

Corrective features present in Modula-2

1. Modula-2 supports dynamic arrays.
2. Variables declared in a library module stay "alive" for the duration of a program's existence. The scope of these variables can be carefully controlled.
3. Case statements can use an "else" clause.

4. Separate compilation with rigorous cross-reference checking is provided by the language.
5. Data hiding is achieved through the module and the opaque type.
6. The declaration order for data objects is totally relaxed.
7. Type checking may be suppressed by using the WORD type.
8. Modula-2 provides complete support for concurrent processes.

Figure 7 A Comparison Between Pascal & Modula-2

2.3 Library Modules

A library module offers a set of named facilities that can be EXPORTed to other modules. A main module, however, can only IMPORT. Furthermore, the source text of a library module consists of two parts : the DEFINITION part which describes the interface offered to other modules, and an IMPLEMENTATION part which contains the code (procedure bodies and variables) necessary to implement the offered interface.

Prior to the execution of the statement following BEGIN in the main module, all other modules which are IMPORTed will have their executable statement part executed. This is because variables declared in such modules have the same lifetime as the main module and are therefore frequently used for storing information needed

between successive calls of procedures in the module. For instance, the "InOut" module needs an initialized file descriptor variable for reading characters from the user terminal. This variable should be initialized by the "InOut" module itself. The linker will ensure that such library modules are initialized themselves before the main program starts. Due to its special role as initialization code, the statements between BEGIN and END of a MODULE are called the initialization part of the module.

One very important aspect of the Modula-2 language system is version control. A definition module is "compiled" into a symbol file which not only contains the definition module in an internal and checked form, but also includes a unique compilation identification (eg. time-stamp). When a DEFINITION MODULE carries a "newer" time-stamp than that of the IMPLEMENTATION MODULE, it implies that the implementation may not be consistent with the DEFINITION MODULE. Thus, a version conflict is spotted. This feature is of indispensable importance especially in large projects.

2.4 The Process Concept of Modula-2

Surprisingly, there are no built-in linguistic constructs for handling concurrency. However, the language

is defined in such a manner that concurrent execution of outermost level procedures is assumed possible, implying that variables declared at the static level (e.g. "x", "y", "i" and "k" of Figure 6) are potentially shared variables. Also, the built-in module SYSTEM exports a PROCESS type and a set of primitives for explicitly saving and restoring the central processor state, as well as primitives for establishing mutual exclusion and for dealing with the interrupt system of the target computer. For instance, the SYSTEM procedure call "TRANSFER(current, new)" saves the current state of the process in "current" and restores a processor state from the PROCESS variable "new". Since Modula-2 was specially designed for a single processor machine, it can attain only so called "quasi-concurrency".

2.5 Type Coercion

Although strongly typed in general, the Modula-2 language has escape mechanisms which allow arbitrary types to be imposed on a particular value. For instance, if T is a type name, the expression T(e), where e is an arbitrary expression, is a valid expression of type T. Also, formal parameters of type ARRAY OF WORD will be compatible with any actual argument type.

2.6 The Prospects of Modula-2

The British Standard Institute is attempting to produce a standard proposal for the language. Because of the magnitude of the job, the proposal was started in 1984 and is expected to be finished in 1988, but so far nothing has been finalized. Nevertheless, the Modula-2 language is gaining its popularity among many educational institutions.

Chapter 3

Run-Time Libraries in C

3.1 Introduction

As discussed in section 1.7, the one-pass Modula-2 compiler for the AT&T UNIX PC's model 3B1 is written in Modula-2. It is first compiled on an AT&T 3B15 and then ported to the UNIX PC. There are six fundamental library modules that the compiler will need and they are implemented in C. Each of the library modules may contain types, variables, constants or procedures.

In any kind of compilation, there must be some files involved, at least the source code. Therefore, the library "FileSystem" is in the run-time libraries to support different operations with files. In order to echo back the status of the compilation, "InOut" and "Terminal" are also included to direct messages generated by the compiler to the standard output and the screen respectively. There are also "Storage" to provide work space, "Arguments" to take the arguments on the command line, and "DateTime" to generate time-stamps for version conflicts checking. The components of each library modules will be discussed in Appendix 1.

3.2 Adapted Features of the C Library Modules

When I wrote and used this set of library modules, some special features I had to be aware of. As it will be stated again in section 6.3, the order of pushing the parameters on the stack in C is not the same as in Modula-2. "CIMPORT" is used in the Modula-2 programs to instruct the compiler to push the parameters in a proper order when importing C routines. The number of bytes for types INTEGER and CARDINAL in Modula-2 does not match with those in C and there is even not a CARDINAL type in C. Then the unsigned integer is used in lieu of CARDINAL. The UNIX PC cross compiler prefixes the imported objects by their library module names and an underscore in the process of code generation. Therefore, in order for the compiler to recognize the imported objects from the C routines, the library module name and an underscore also have to be added to the object names whenever an imported object is used. These feature are temporarily adapted to support the execution of the UNIX PC cross compiler. Once the UNIX PC Modula-2 compiler is created, then all the library modules can be written in Modula-2 and the normal Modula-2 syntax can be resumed.

Chapter 4 Coroutine Primitives

4.1 Introduction

NEWPROCESS and TRANSFER are procedures from a Modula-2 library called SYSTEM. They are the primitives for concurrent programming in Modula-2. Since these two procedures involve manipulating the registers, they are implemented in Assembly (MC68000 microprocessor). Before each of the coroutine primitives is discussed, let's familiarize ourselves with the microprocessors that we will be dealing with.

4.2 Architectural Overview

NEWPROCESS and TRANSFER are implemented in Assembly language (MC68010 16-/32-bit microprocessor). The corresponding implementation (for VAX Central Processor) on DEC VAX 11/780 was taken as a reference.

Although the MC68010 microprocessor and VAX Central Processor are two different processors, the essence of the low-level process operations is based on some common grounds. The following sections serve to elucidate succinctly and yet thoroughly these fundamental concepts of concurrent programming.

4.3 UNIX PC Microprocessor

The AT&T UNIX PC uses a MC68010 16-/32-Bit Virtual Memory Microprocessor (Reference number ADI-942) introduced by Motorola in 1979. (M68000 is used hereafter to refer to the M68000 16-/32-bit microprocessor architecture of which MC68010 is an implementation.)

The M68000 executes instructions in one of two modes--user mode or supervisor mode. The user mode is intended to provide the execution environment for the majority of application programs. The supervisor mode allows some additional instructions and privileges and is intended for use by the operating system and other system software.

As shown in the user programmer's model in Figure 8, the M68000 offers sixteen 32-bit general purpose registers (D0-D7, A0-A7), a 32-bit program counter, and an 8-bit condition code register. The first eight registers (D0-D7) are used as data registers for byte (8-bit), word (16-bit), and long word (32-bit) operations. The second set of seven registers (A0-A6) and the stack pointer (USP) may be used as software stack pointers and base address registers. In addition, the address registers may be used for word and long word operations. All of the sixteen registers may be used as index registers.

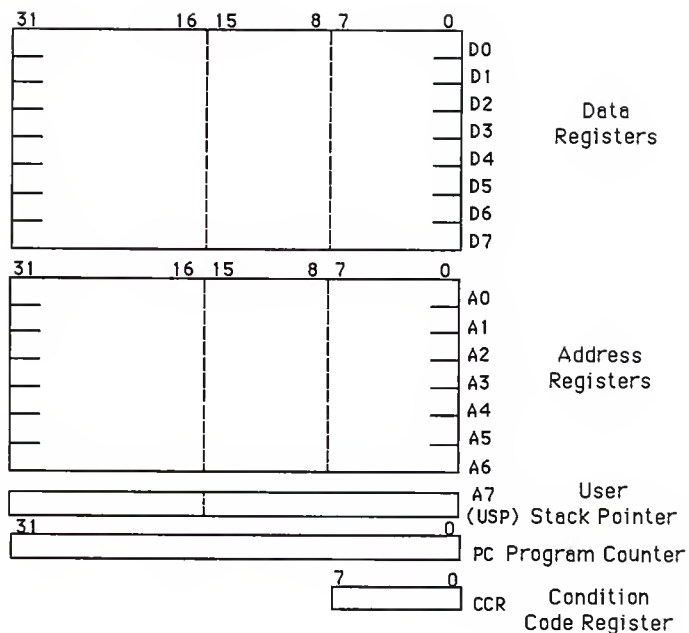


Figure 8 Programmer's Model of MC68010

In most systems using the MC68010 as the central processor, such as the AT&T UNIX PC, only a fraction of the address space will actually contain physical memory. However, by using virtual memory techniques the system can be made to appear to the user to have a large amount of physical memory available.

The basic mechanism for supporting virtual memory in computers is to provide only a limited amount of high-speed physical memory that can be accessed directly by the processor while maintaining an image of a much larger virtual memory on secondary storage devices such as large capacity disk drives. When the processor attempts to access a location in the virtual memory map that is not currently residing in physical memory (referred to as page fault), the access to that location is temporarily suspended while the necessary data is fetched from the secondary storage, and placed in physical memory; the suspended access is then completed. The MC68010 provides hardware support for virtual memory with the capability of suspending an instruction's execution when a bus error is signaled and then completing the instruction after the physical memory has been updated as necessary.

4.4 VAX Central Processor

Since NEWPROCESS and TRANSFER are modeled after the implementation for DEC VAX 11/780 central processor, we will be focusing on only the issues related to concurrent programming on the VAX such as processes, registers, and the stack.

VAX architecture is intended to support multiprogramming, the concurrent execution of a number of processes in a single computer system. (A process can be defined for now as a single stream of machine instructions executed in sequence.)

A VAX process exists in and operates on a memory space of four Mega bytes; certain addresses and data are kept in the sixteen 32-bit general registers; and a small number of processor state variables are kept in a special register called the Processor State Longword, or PSL. The combined set of information in memory, general registers, and PSL actually defines a process.

The VAX provides sixteen general registers for temporary address and data storage. Registers do not have memory addresses, but are accessed either explicitly by inclusion of the register number in an operand specifier, or implicitly by machine operations which make reference to specific registers. Certain registers have specific uses and special names:

PC R15 is the Program Counter (PC). The processor updates it to address the next byte of the program; therefore, PC is not used as a temporary, accumulator, or index register.

SP R14 is the Stack Pointer (SP). Several instructions make implicit references to SP, and most software assumes that SP points to memory set aside for use as a stack. There is no restriction on the explicit use of other registers (except PC) as stack pointers, though those instructions which make implicit references to the stack always use SP.

FP R13 is the Frame Pointer (FP). The VAX procedure call convention builds a data structure on the stack called a stack frame. The CALL instructions load FP with the base address of the stack frame, and the RETurn instruction depends on FP's containing the address of a stack frame. Further, VAX software depends on maintenance of FP for correct reporting of certain exceptional conditions.

AP R12 is the Argument Pointer (AP). The VAX procedure call convention uses a data structure called an argument list, and needs AP as the base address of the argument list. The CALL instructions load AP in accordance with that convention, but there is no

hardware or software restriction on the use of AP for other purpose.

R6:R11 Registers 6 through 11 have no special significance either to hardware or the operating system. Specific software will assign specific uses for each register.

R0:R5 Registers 0 through 5 are generally available for any use by software, but are also loaded with specific values by those instructions whose execution must be interruptible--the character string, decimal arithmetic, Cyclic Redundancy Check, and Polynomial instructions. The specific instruction descriptions identify which registers are used, and what values are loaded into them.

A stack is implemented in the VAX by a block of memory and a general register which addresses the "top" of the stack. The "top" of the stack is that location in the block which contains the next candidate for removal. An item is added to the stack ("pushed on") by decrementing the register which serves as the stack pointer, and storing the item at the address in the updated register. The pointer is decremented by the length of the item added to the stack, to allow enough room for it. Conversely, the top item is removed ("popped off") by adding the length of

the item to the stack pointer after the last use of the item.

4.5 Functional Specifications

```
NEWPROCESS (p: PROC; wkspadr: ADDRESS;  
            wkssize: CARDINAL; VAR cor: PROCESS);
```

This procedure creates a new process. Four parameters are needed in order to call this procedure. "p" is a parameterless procedure used as the body of the coroutine. This procedure contains the statements executed by the coroutine. "wkspadr" is the address of the start of a block previously allocated memory used for the work space of the coroutine. This block will contain the coroutine's local variables and its state of execution while it is suspended. "wkssize" is the size, in bytes, of the pre-allocated block of space which starts at "wkspadr". The larger the size and number of local variables in the coroutine are, the larger this parameter should be. PROCESS "cor" is the new initialized coroutine. It is initialized in a state such that when control is transferred to it the first time, execution starts at the beginning of procedure "p". The coroutine will remain in existence until the entire program terminates.

The way this procedure works is shown in Figure 9.

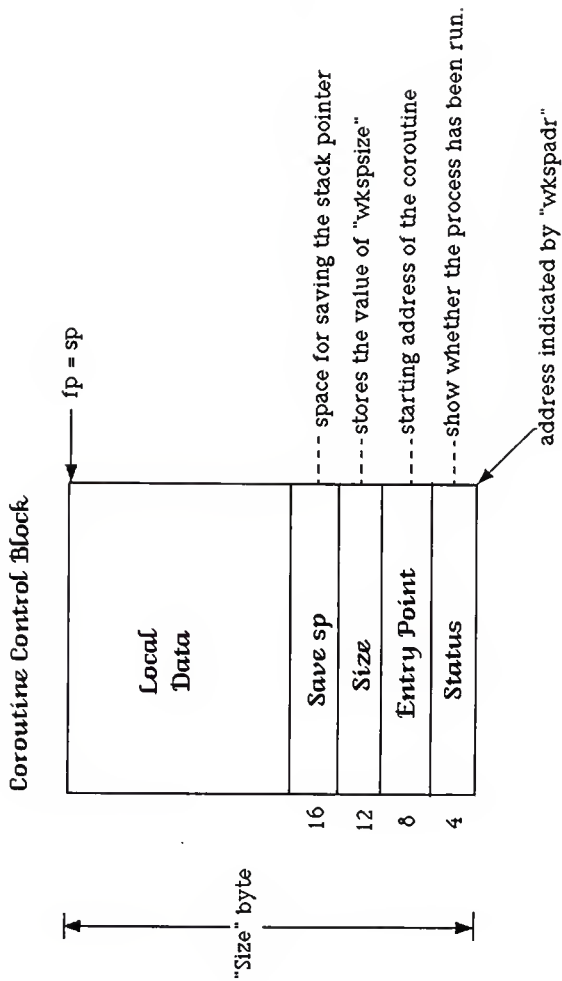


Figure 9 Coroutine Control Block. Created by NEWPROCESS

Therefore, the following paragraph should be read in conjunction with the figure. First, the address that "wkspadr" carries is moved to a register, r1. "wkspsize" is subtracted from r1 to designate a block of memory for the new process called coroutine control block or ccb and meanwhile, move the stack pointer to the end of the stack. The first byte of the block, "status", is turned off to indicate the process has not started yet. The entry point of the process and the size of the stack are stored in the second and third byte of the block respectively. The content of r1, which is the address of the end of the stack, is returned in "cor". Since all the necessary information to execute the coroutine is captured in the ccb, a new process is thus created. It can be referred to as "cor" later on.

TRANSFER (VAR source, destination: PROCESS);

When this procedure is called, coroutine "source" will be suspended and coroutine "destination" will be activated. The first time a coroutine gets control in this manner, it starts at the beginning of the parameterless procedure assigned to it. Subsequent transfers of control to that coroutine result in execution resuming with the first statement after the TRANSFER call that caused it to be suspended.

Procedure TRANSFER has a slightly more complex mechanism in its implementation than NEWPROCESS. Figure 10 shows the modifications made on the coroutine control blocks in the course of transferring control from process "source" to process "destination".

Once TRANSFER has been invoked, the return address of the executing coroutine is saved in its ccb. Then the current states of all the registers are pushed into the stack so that the run-time environment can be restored when the control is switched back to coroutine "source". The register, r1, is designated to contain the ccb address for the current running process; whereas register r2 is made to point to coroutine "destination". The current stack pointer is stored in "savesp" of "source's" ccb.

After modifying "source's" ccb, the new process "destination" gets ready to run. If the status shows the process has not been executed before, the status byte is clear, then the status will be set to 1. After getting the entry point address of the coroutine and setting the frame pointer and the stack pointer to the stack area of the new process, coroutine "destination" is called. The control will not return until the procedure that "destination" represents terminates. The whole program will also terminate thereafter.

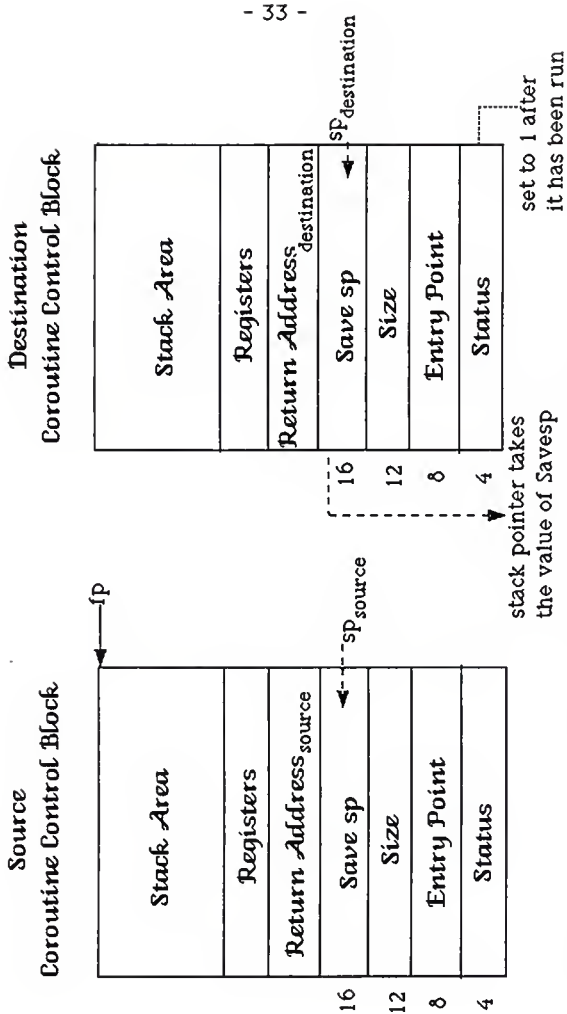


Figure 10 The Mechanism of Transfer (source, destination)

However, if the status byte shows that the new process has been run before, the stack pointer will be assigned to where "saveesp" indicates. In fact, it is the statement right after the last TRANSFER call from coroutine "destination". All the registers previously pushed into the stack are popped off now and the previous runtime environment is resumed. Therefore, the new process can continue its execution. At the same time, control is returned to the process representing "source" and it begins executing the rest of its code.

4.6 Conventional Differences in Implementations

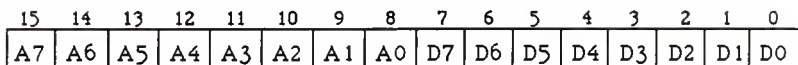
A number of differences in conventions between these two processors were encountered and resolved in the course of implementing the coroutine primitives.

For the VAX Central Processor, registers are named "rn" where n is an integer from 0 to 15. A table of the VAX Central Processor registers special usage is shown in Figure 11. MC68010, however, has two categories of registers, namely data registers (d0 to d7) and address registers (a0 to a7). Unlike the register set of VAX Central Processor, there is not a register dedicated to be an Argument Pointer. The offset of the Frame Pointer is used to find the addresses of the arguments in MC68010.

Register	Name	Assembly Syntax	Assigned Function
0	r0	%r0	Results of functions, status of services
1	r1	%r1	Results of functions
2, 4	r2, r4	%r2, %r4	Length counter in character & decimal instructions
3, 5	r3, r5	%r3, %r5	Address counter in character & decimal instructions
6 - 11	r6 - r11	%6 - %11	General-purpose
12	AP	%ap or %r12	Argument pointer
13	FP	%fp or %r13	Frame pointer
14	SP	%sp or %r14	Stack pointer
15	PC	%pc or %r15	Program Counter

Figure 11 The Register Set of VAX Central Processor

Postincrement Addressing Mode



Predecrement Addressing Mode

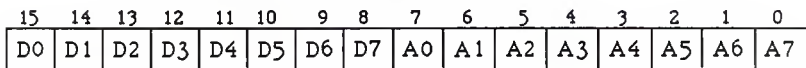


Figure 12 Two Addressing Modes of MC68000

Note that the Frame Pointer and the Stack Pointer are located at registers 13 and 14 in Figure 11. In MC68010, the corresponding registers reside in a7 and a6 respectively. Moreover, the configuration of the MC68010 registers appears to be reversed in the predecrement addressing mode and the postincrement addressing mode (Refer to Figure 12) when registers are to be transferred or masked. Registers are pushed on the stack and the stack pointer is decremented--predecrement addressing mode--and popped out before the stack pointer is incremented--postincrement addressing mode. However, masking the registers does not alter the configuration of registers on VAX.

Furthermore, the arguments are pushed onto the stack in just an opposite order in these two processors. On VAX, the first argument is pushed down onto the stack first; however, MC68010 does the opposite, the last argument gets pushed down first.

Since Assembly Language is very machine dependent, variations in syntax among different processors are inevitable. The syntax of MC68010 and VAX Central Processor diverse in size specification, global declaration, address indirection, and some other individual commands. Examples are listed in Figure 13.

<u>VAX Processor</u>	<u>MC68010</u>	<u>Description</u>
calls \$n, <ea>	jsr <ea>	Call a subroutine
jeql	beq.b	Jump when equal
pushr	movm.l <<>, -(%sp)	Push registers
popr	movm.l (%sp)+, <<>	Pop registers
ret	rts	Return from subroutine
movl	mov.l	Size specification (l vs .l)
mova	lea	load address
*n(ap)	2 mov statements	Address indirection
\$	&	Value sign

Figure 13 Examples of VAX Central Processor
and MC68010 Syntax

Chapter 5

The Standardization of Modula-2 Libraries

The British Standard Institute (BSI) has not come to a consensus on a standard proposal for the Modula-2 libraries at the time this report is written. Nevertheless, some articles and personal attendum have been published to voice their opinions on the standards.

The standard I/O libraries and the utility libraries from BSI are discussed so that we can gain a wholesome perspective on standardization of Modula-2 libraries.

5.1 Draft BSI Standard I/O Library of Modula-2

In their interim report on the progress of the standard I/O library, the following three properties were specified as their design goals :

- (i) it should be layered to ease implementation and to allow different types of user access to features and facilities of varying levels of power and sophistication;
- (ii) it should be extensible so that facilities for user-defined data types and device-specific drivers can be added at will;

(iii) it should be robust so that ordinary high-level language programmers can have error-free and crash-free access to the I/O devices without having to resort to doing their own parsing.

5.2 Layering

The current state of the design sees three layers : the bottom, middle, and top layer.

The bottom level contains the device-specific driver modules. There must be a module for each device to be accessed through a Modula-2 program.

The middle level implements a buffering system between the lower level device drivers and the higher level type I/O modules. This level provides a character/word stream to any module at the top level.

The top level contains the type-specific modules. They are modules of the same set of operations for each standard Modula-2 simple type which are BOOLEAN, CARDINAL, INTEGER, REAL, and CHAR. They are named after the corresponding type with a suffix "IO" attached to it in the standard I/O library. Each component of the standard set of operations will be discussed individually in a later section.

5.3 Extensibility

When data of a specific type is used, the relevant I/O modules must be imported. When a new type is defined, the programmer should be able to create a similar module to handle that type. Therefore, I/O operations among various types are standardized.

The I/O library can serve four classes of users :

- (1) Those writing applications using standard types (top level users);
- (2) Those creating new types which are the more efficient implementations of the I/O operations (middle level users);
- (3) Those controlling devices directly (bottom level users);
- (4) Those interfacing new devices to the system (extended bottom level users).

5.4 Robustness

This design makes use of predicates to pretest whether a specific input operation will succeed before the action really takes place. Therefore, it frees us from the worries of crashing the system by some invalid inputs.

If the I/O operation is tested unsuccessful, the programmer can take the corrective action accordingly.

5.5 The Semantics of the Top Level Interface

The top level interface deals with the input and output of the five Modula-2 base types : CHAR, CARDINAL, INTEGER, BOOLEAN and REAL. All input and output is performed via explicitly referenced channels, supported by the middle level interface that are connected to any device. Each of the five types have their own I/O module, all of which export seven procedures. These procedures are identical in semantics but as they deal with different objects have slightly different syntax. The definition modules can be found in Appendix 2. The opaque type "Channel" is imported from the middle level.

5.6 Channels

All channels use blocking reads; that is, if no data is currently available from the device attached to the channel, a blocking read is caused and the procedure pauses until data is available. The channels can be opened as binary and character, so the implementation of the object I/O modules will ensure that the object is written in a binary form.

Following is a discussion of the seven operations in the I/O modules.

CanSkip

This predicate function returns TRUE if an object is available from the channel specified and FALSE when the channel has reached the end of the stream.

CanRead

This predicate function returns TRUE if an object is available from the channel specified and it can be represented internally. On the contrary, if the value of the object represented by the stream associated with the channel cannot be represented within the computer's hardware or the channel has reached its end-of-stream condition, FALSE is returned.

Skip

This procedure will skip over the current object on the channel, thus changing the state of the channel.

Value

This procedure will return the next available object from the specified channel.

Read

This procedure will return in its second parameter the current object on the channel.

Write

This procedure will write its parameter to the channel.

Print

This procedure will write its parameter to the channel in a formatted way. The length that the object is printed in is governed by the value of the third parameter. Optional right-justification is provided. Blanks will be padded at either end according to the right justified flag if the object does not occupy all the spaces provided.

5.7 Draft BSI Standard Utility Library of Modula-2

A set of goals is specified by BSI to govern the directions of the efforts in standardizing the utility library modules. They are listed as follows :

- (1) allows one to write portable Modula-2 programs which may be moved between implementations;

- (2) provide facilities for writing software tools (eg. cross referencers, compilers, prettyprinters, ...) and "data processing" applications; but not graphic editors, communications programs and so on;
- (3) allow one to describe algorithms which refer to a standard environment;
- (4) do not specify the operating system (eg. tasking, device IO, exceptions, ...);
- (5) provide Pascal-like facility for the naive user;
- (6) offer the experienced programmer more control;
- (7) do not preclude addition of environment dependent features (eg. tasking, exceptions, ...);
- (8) do not attempt to provide portability of data files between implementations or environments;
- (9) provide the facilities of Pascal and C with library but not all those of Ada.

It is apparent that the principles of designing the utility library do not deviate very far from those of the I/O library. The I/O standard library provides a set of I/O primitives on which the utility library can be built. Only the I/O standard library is intimately related to the

machine architecture; whereas the utility library can be freed from the hardware dependency. Hence, the portability of Modula-2 programs between implementations can be achieved.

Appendix 3 is the draft utility library of Modula-2. In comparison with the library modules implemented in this project shown in Appendix 4, one may find a more collective grouping of library modules in our implementation. For instance, the BSI MODULES Files, Binary, Directory, Text, and NumberIO are all included in FileSystem in our implementation. In so doing, all the operations that deal with files can be in one library module. It is easier to manage, especially when a large number of library modules are involved. The user will also find it more convenient to keep track of the library modules from which the objects are IMPORTed.

The utility library of our implementation can satisfy a wider spectrum of needs than those in Modula-2 library draft of BSI (Appendix 3) because almost all the objects specified in the draft of BSI can be found in our implementation. In addition, library modules Arguments, Date-Time, Processes, and Coroutines are also introduced to facilitate operating system and concurrent programming. Although the goal statement (4) can justify for BSI not

having the operating system library modules, there is still some uncertainty they need to resolve (Refer to Appendix 3, DEFINITION MODULE SYSTEMToBeImpl). Because of the flexibility of the language and the diversity of pragmatic requirements from different users, a consensus on the standardization of Modula-2 library will be difficult to arrive at.

Chapter 6

Modula-2 Libraries on the UNIX PC

6.1 Introduction

Modula-2 was designed to be an open ended language. Specific usages are left for the users to tailor it to their own needs. On this point, Niklaus Wirth, the Father of the Modula-2 language, commented, "... a language standard should only guarantee program behavior where the language features are used in a way that is consistent with the overall design intent and good programming practice, leaving other usage in some undefined or unstandardized state." [5] Therefore, the library modules included in this project mainly are for pragmatic reasons: research and teaching purposes.

Some existing Modula-2 library modules have been proven effective in an academic environment. In the process of implementing the library modules for this project, the decision on which modules should be included was based on three key references. Niklaus Wirth covers practically all facilities of the language in the book "Programming in

[5] Welsh, Jim & Bailes, Paul. "Modula-2 Standardisation: The Go-Betweens' Tale", The MODUS Quarterly, issue #7, February 1987, p 4.

Modula-2" [6]. We try not to deviate very far from the modules specified in the book. The second source was the 3B/System V implementation of Modula-2 utility modules from Brussels Free University [7]. The implementation of the Coroutine primitives on the VAX 11/780 [8] was also examined for the underlying coding for the quasi-concurrency mechanisms in Modula-2.

Besides the references mentioned above, some new facilities are also introduced in this implementation.

Three different computer languages are used to implement various functionalities of the Modula-2 language on the UNIX PC. They are Assembly (16-/32-bit MC68010), C (on AT&T 3B15 and AT&T UNIX PC Model 3B1), and Modula-2 (on AT&T UNIX PC model 3B1).

6.2 The Implementation in Assembly

The assembler in the UNIX PC is for the 16-/32-bit MC68010 microprocessor which has 32-bit registers and a

[6] "Programming in Modula-2" by Niklaus Wirth, third edition, Springer-Verlag International, Inc.

[7] The authors of this implementation are Rudi Leonard, Thomas Rottenberg, Yves Vandenbosch, and Benoit Van Hove from Brussels Free University, Faculty of Applied Sciences, Informatics Department, Pleinlaan 2, 1050 Brussels, Belgium.

[8] This implementation is from Western Research Laboratory of Digital Equipment Corporation on DEC VAX 11/780 and is to support the Modula-2 compiler "mod2".

16-bit data bus. Unlike the microprocessors with a 32-bit data bus, it takes MC68010 two data transmissions to get a full register. Therefore, the coding of the 32-bit signed and unsigned multiplication and division, and the 32-bit and 64-bit arithmetics have to take this issue into account.

The Coroutine primitives NEWPROCESS and TRANSFER are implemented in Assembly of MC68010. The parameters are pushed on the stack in the order they appear in the parameter list. It is just the opposite of the way that the C processor handles parameters. In our implementation of Coroutine primitives, pushing the parameters, for instance a procedure call $x(a,b)$, onto the stack is implemented in the fashion as shown in figure 14 and 15. The other aspects of implementing the Coroutine primitives are discussed in Chapter 4.

```
push a
push b
bsr x
link fp, #num           where "num" is the size of the
                        local data for procedure x.
```

Figure 14 The Assembly Instructions of Pushing Parameters on the Stack

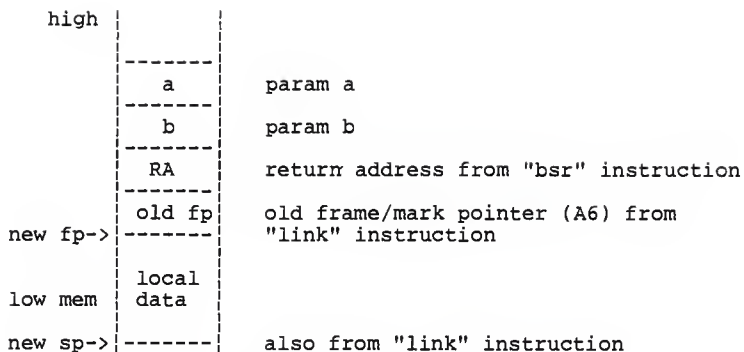


Figure 15 The Stack Configuration of MC68010

6.3 The Implementation in C

The language C is used to write the run-time libraries for the Modula-2 compiler on the UNIX PC. The run-time libraries provide the implementation of the basic functions that the Modula-2 compiler calls when it is ported to the UNIX PC. The details are discussed in Chapter 1. However, it should be noted that a new reserved word, "CIMPORT", is introduced in this project. When a Modula-2 program needs to "IMPORT" a C function, it has to be included in the declaration section of the program. In order to inform the compiler that the parameters are to be pushed on the stack in a reversed order, "CIMPORT" is created to serve this purpose.

6.4 The Implementation in Modula-2

The Coroutine primitives are implemented in Assembly, while the rest of the utility library modules are implemented in Modula-2. Most of them are specified either in Wirth's book, "Programming in Modula-2", or in the implementation from Brussels Free University. Some new ones are introduced to make this implementation of Modula-2 utility libraries a more powerful one.

Being a descendent of Pascal, Modula-2 also did not have any library functions to operate on strings except that the programmer has to deal with them one array cell at a time. Seeing the demand for operations on strings, `StringLen` and `CmpString` are introduced in addition to `CopyString` and `AppendString`. `StringLen` returns the length of the string without counting the null character whereas `CmpString` compares two strings and returns an integer value to indicate the lexical difference of two strings.

In Modula-2, a CASE statement can be used in a very unique way that few other computer languages are capable of duplicating the same function with their corresponding constructs. The expression that a Modula-2 CASE evaluates does not have to be an expression! This paradoxical facility can support a more flexible usage of a CASE statement in Modula-2. If just a type, for instance

CARDINAL, replaces an expression, any option within the CASE statement can be taken. Note that all the options, however, operate on the same memory location. The following illustration may shed some light on this special functionality.

Consider a RECORD, Template, in figure 16. It occupies four bytes of memory. However, we can operate the variables of type Template with any variables of type ranging from BYTE, WORD, to LONGCARD.

```
TYPE
  Template = RECORD
    CASE CARDINAL OF
      0 : b1, b2, b3, b4 : BYTE
      | 1 : w1, w2       : WORD
      | 2 : lc          : LONGCARD
    END
  END;
```

Figure 16 A Modula-2 CASE Statement

For instance, Mode is of type Template. The following assignment statements in figure 17 are all valid. If "AB" is passed in word1 in figure 17, then Mode will be holding "ABBA" at the end of the assignment statements.

```
PROCEDURE MirrorImage ( word1 : WORD );
VAR
  Mode : Template;

BEGIN
  .....

  WITH Mode DO
    w1 := word1;
    b3 := b2;
    b4 := b1;
  END;
  .....

END MirrorImage;
```

Figure 17 An Application of CASE Statement

In Wirth's book [p.72], the syntax of CASE statement expression is as follow :

CASE variable : type OF

In our implementation, the colon, ":" [9], is required only when the CASE statement is used with a non-deterministic branching as in the example shown in figure 16. The reserved word "CASE" followed by a colon, ":", indicates that the variable in the expression is omitted. Therefore, branching does not depend on the value of the CASE statement labels, but rather the contents after the labels. Referring back to figure 16, the CASE statement labels, 0, 1, and 2, have no significance in determining

[9] The implementation from Brussels Free University, ":" is not necessary.

whether Template should be a concatenation of four bytes or two words. The invocations of b1 to b4, w1 to w2, or lc govern the constituents of Template. This technique is employed in the implementation of WriteByte, WriteShortWord, and WriteWord of FileSystem.

In a deterministic CASE statement, the variable in the expression carries a value which is to match the CASE statement labels and execute the statements thereof. The type of the variable can be retrieved from the symbol table, so the colon, ":", and type in the expression are omitted in our implementation.

Wirth includes a mathematics library, MathLib0, in the set of standard utility modules. The facilities involved are some fundamental mathematical operations such as square root, sine, cosine, logarithm, and so on. This math library is general and yet inclusive. Before implementing this library module, I had studied some other implementations for personal computers running MS-DOS operating system written in Modula-2 and for VAX 11/780 written in C. The one for MS-DOS computers is too machine dependent. There are many architectural differences that cannot be reconciled. Consequently, the C implementation on the VAX was studied in depth.

Cody & Waite's algorithms and coefficients for sine,

cosine, logarithm, and exponent [10] are adopted in our implementation; whereas Newton's Method for Estimating Zero of Function is used for computing the square root of a real number. Being limited to 32-bit calculations, precision up to seven significant digits is the best accuracy that can be generated by this Modula-2 compiler. Discrepancies become eminent when the numbers grow beyond seven significant digits.

Besides "CIMPORT" we introduced earlier in section 6.3, a change has been made to the "EXPORT" clause in the DEFINITION MODULE also. In our implementation, any objects declared in the DEFINITION MODULE are automatically visible to other modules. Compiling a DEFINITION MODULE produces a symbol file in which the declarations of the objects are located. Whenever an object is called from an external module, the compiler will search the symbol files indicated by the IMPORT statements to find the declaration of that object. In other words, everything declared in the DEFINITION MODULE is made accessible by the compiler through an IMPORT statement. Therefore, an "EXPORT" clause is no longer necessary. However, a higher level of data abstraction for the objects CONST, TYPE, and

[10] Cody, W.J. et al, "A Proposal Radix- and Word-length-independent Standard for Floating-point Arithmetic", MICRO, August 1984, IEEE.

VAR, has to be sacrificed because there is no "hidden" declaration for these three kinds of objects. Being a one-pass compiler, it has some trade-off between compilation speed and level of data abstraction. Figure 18 displays the formats of these two types of data abstraction.

DEFINITION MODULE A;	DEFINITION MODULE B;
TYPE	TYPE
X; (* X's declaration is hidden. *)	X = Y; (* X's declaration is
Illegal declaration in our implementation.	Valid statement in our implementation.

Figure 18 Levels of Data Abstraction

The definition of a WORD of memory is very machine dependent. For the UNIX PC, a WORD is equivalent to two bytes. In some Modula-2 implementations where a WORD is four bytes, a SHORTWORD is used to represent two bytes of memory. In our implementation, there are no SHORTWORDS.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this project, three different computer languages were used to implement seventeen different library modules. They are C, Assembly of microprocessor MC68010, and Modula-2. Among all the library modules, I implemented totally three hundred and forty-seven exported objects which comprise sixty constants, fourteen variables, thirty types, and two hundred and forty-three procedures.

Modula-2 is a computer programming language with great potential. It is sufficient to support many applications. Big tasks such as real-time programming and operating system programming can be entrusted in the hands of a Modula-2 programmer; also as a teaching tool, Modula-2 will not fail to lay a solid foundation for beginning programmers. However, the versatility of the language mainly comes from the capability to allow library modules written to meet specific needs. While everybody is writing their own library modules, the issue of standardization of library is professing with the intent to unify the efforts of language development and provide a set of

standard functionalities for the language. It is anticipated that more computer scientists will endeavor to improve Modula-2 as they realize the significance of the language in the realm of Computer Science.

7.2 Future Work

In this project, we did not concern ourselves with the graphics facilities of Modula-2. In fact, both the Modula-2 language and UNIX PC are capable of supporting these functionalities. In "Programming in Modula-2", Wirth also includes five library modules for window operations. Last year (1987), attempts were made to do graphics programming in C on the UNIX PC. Dealing with a new product, people often fell into seemingly unprecedented problems with the UNIX PC. Experiences and knowledge accumulate as time passes and a better understanding of the UNIX PC is attained. Implementing the graphics library modules should be a less painstaking job than before, but it is still an enormous task.

The inconsistency in size for the same type may hinder the portability of the language. INTEGER and CARDINAL are both two bytes in this implementation, so calculations involved more than five digits have to involve numbers of type LONGINT or LONGCARD which occupy four

bytes of memory. On the other hand, integer in C is four bytes. Therefore, calling C functions from a Modula-2 program should alert the programmer to carefully match the types of the parameters and the return values. In essence, having two different types for the same objects but just with different upper bounds easily frustrate many careless programmers with a series of type clashing and related errors. Aggravation can be alleviated if INTEGER and CARDINAL in Modula-2 are made four bytes also.

The interim report from the British Standard Institute on the progress of the standard I/O library states the needs and advantages of having a standard library for I/O primitives which are discussed in Chapter 5. If all the I/O operations are built from these primitives, the utility library modules can be independent of the computer architecture and become very portable. Since only the standard I/O library primitives have to be changed when porting to another machine, the rest of the utility library modules can remain intact.

There is another proposal in its experimental stage. The Australian computer scientist, George M. Mohay, is attempting to simplify the interface which deals with interrupts and do away with IOTRANSFER. He also claims to be able to modify TRANSFER from taking four parameters

down to one for a PDP-11 Modula-2 system. Only part of his proposal has been implemented. However, it would be convenient if these facilities are also available in our UNIX PC Modula-2 system.

Bibliography

Papers:

- (1) Bilbe, Charles R, "Using the Heap for Modula-2 Opaque Types", Journal of Pascal, Ada, & Modula-2, November/December 1985, pp 24-30.
- (2) Bondy, Jon, "Modula-2 Standard Library Documentation", Modula-2 News, issue #1, January 1985, pp 38 - 58.
- (3) Bush, Randy, "Modula-2 Standard Library Rationale", Modula-2 News, issue #1, January 1985, pp 20 - 21.
- (4) Djavaheri, Morris, "An Implementation of the Proposed Standard Library for the UNIX Operating System", MODUS Quaterly, issue #4, November 1985, pp 26-27.
- (5) Djavaheri, Morris & Osborne, Stan, "Modula-2: An Alternative to C for System Programming", Journal of Pascal, Ada, & Modula-2, May/June 1986, pp 47-51.
- (6) Eisenbach, Susan, "Draft BSI Standard I/O Library for Modula-2", MODUS Quaterly, issue #5, February 1986, pp 15-23.
- (7) Mazlack, Larry, "Academic Modula-2 Survey", The MODUS Quarterly, issue #6, November 1986, pp 37 - 41.
- (8) Mohay, George M., "A Simplified Coroutine Structure for Modula-2", Journal of Pascal, Ada, & Modula-2, January/February 1987, pp 35-42.
- (9) Odersky, Martin et al, "Proposal for a Standard Library and an Extension to Modula-2", MODUS Quaterly, issue #4, November 1985, pp 13 - 25.
- (10) Olsen, Peter, "Modula-2 for Real-time Systems", Computer Standards & Interfaces, vol 6, November 1987, North-Holland, pp 61-70.
- (11) Torbett, Micheal A., "More Ambiguities and

Insecurities in Modula-2", SigPlan Notices, vol 22 #5, May 1987, pp 11-17.

- (12) Ward, Don, "BSI Modula-2 Working Group Standard Concurrent Programming Facilities", The MODUS Quarterly, issue #8, May 1987, pp 35 - 60.
- (13) Welsh, Jim & Bailes, Paul, "Modula-2 Standardisation: The Go-Betweens' Tale", The MODUS Quarterly, issue #7, February 1987, pp 3 - 6.

Books or Magazines:

- (14) AT&T, "WE 32100 Microprocessor Information Manual", The AT&T Documentation Management Organization, Issue 2, November 1986.
- (15) Bach, J. Mauric, "The Design of the UNIX Operation System", Prentice-Hall International Editions, 1986.
- (16) Digital, "VAX Architecture Handbook", Digital Equipment Corporation 1981.
- (17) Gleaves, Richard, "Modula-2 for Pascal Programmers", Spring-Verlag, New York, Inc., 1984.
- (18) Kernighan, Brian W. & Ritchie, Dennis M., "The C Programming Language", Bell Telephone Laboratories, Inc., 1978.
- (19) Moore, John B. & McKay, Kenneth N., "Modula-2, Text and Reference", Prentice-Hall, Inc., 1987.
- (20) Motorola, "M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual", Prentice-Hall, Englewood Cliffs, NJ, 5th edition, 1986.
- (21) Munem, M.A. & Foulis, D.J., "Calculus with Analytic Geometry", Worth Publishers, Inc., 2nd edition, 1984.
- (22) Sale, Arthur, "Modula-2 Discipline & Design", Addison-Wesley Publishing Company, Inc., 1986.
- (23) Smedeam, C. H. et al, "The Programming Languages", Prentice-Hall International, Inc., 1983.
- (24) Tondo, Clovis L. & Gimpel, Scott E., "The C Answer Book", Prentice-Hall, Inc., 1985.

- (25) Wiener, Richard & Sincovec, Richard, "Software Engineering with Modula-2 and Ada", John Wiley & Sons, Inc., 1984.
- (26) Wirth, Niklaus, "Programming in Modula-2", Springer-Verlag, 3rd corrected edition, 1985.

Electronic Mail:

- (27) alan@pdn, "Re: Procedure Parameters and Transfer", from alan@pdn to ksuvax1.KSU, 16 September 1987, Alan Lovejoy.
- (28) cbilbe@Sun.COM, "Coroutine vs Operating Systems", from cbilbe@Sun.COM to KSUVAX1.BITNET, 20 November 1986, Chuck Bilbe, Project Leader Modula-2, Sun Microsystems, Inc., Mountain View, CA.
- (29) garon@gr.utah.edu, "Re: Code for thought", from garon@gr.utah.edu to ksuvax1.cis.ksu.edu, 15 November 1987, Garon C. Yoakum.
- (30) joel@decwrl.dec.com, "Re: NEWPROCESS", from joel@decwrl.dec.com to ksuvax1.KSU, 18 September 1987, Joel McCormack.
- (31) rlc@uvacs.cs.virginia.edu, "Procedure Parameters and Transfer", from rlc@uvacs.cs.virginia.edu to ksuvax1.KSU, 13 September 1987, Robert L. Chase, Director of Academic Computing, Compute Center, Sweet Briar College, Sweet Briar, VA.

APPENDIX 1

Overview of C Library Routines

Arguments

There are only one variable and one procedure chosen to be included in the run-time libraries.

ArgCount

It is a variable of type `CARDINAL` and represents the number of command line arguments, including the name of the command invoked (which is always argument 0). Argument number 1 is therefore the first argument to the command, and argument "`ArgCount - 1`" is the last.

GetArgument

It is a procedure which takes in an argument number and returns the corresponding argument and its length.

DateTime

Like "Arguments", there are only two constituents in this library module, namely a type and a procedure.

TimeType

`TimeType` is a predefined `LONGINT` type.

Time

Procedure Time returns current wall clock time in system format, which is the number of seconds since 00:00:00 GMT, January 1, 1970.

FileSystem

It is the most heavily used library module which contains two types and fifteen procedures. They deal with the interface for general disk file I/O.

Close

This procedure closes a file and all the modifications thereof become permanent.

Create

"Create" creates a new file and opens it for read and write access. A File type variable is returned for subsequent I/O.

CreateTempfile

Create a new file and open it for read and write access. The file name passed in should look like a file name with six trailing Xs. These Xs are replaced by a letter and the process ID to make a unique temporary file name.

Delete

The file pointed by the file pointer which is passed into the this procedure will be deleted.

Done

This procedure reports whether or not the previous call to another FileSystem procedure was successful.

File

A file is identified by an external and internal name; the external name is a string according to UNIX filename conventions, the internal name is a Modula variable of type "File" whose structure is hidden from the outside modules.

GetPos

The current position of the file concerned is returned through a variable of type CARDINAL.

ModeType

It is an enumerated type with elements : O_RDONLY, O_WRONLY, O_APPEND, O_RDWR, O_WRONLY, O_WRONLY, and O_APPENDUPDATE. They represent read-only, write-only, append mode, read-write, write-update, and append-update respectively. In the C implementation, ModeType is just a type of unsigned integer and all the constituents carry a different integral values.

Open

A file specified in the "filename" parameter is opened with the access rights indicated by the "mode" parameter.

ReadChar

"ReadChar" reads the character at the current file position and returns it. The file position is then advanced by one byte.

ReadWord

A word is read from the current file position which in turn is advanced by one word (four bytes).

SetPos

The file position is set to byte "position" which is a parameter passed in through SetPos.

ShowStatus

This procedure should be called if "Done" returns FALSE after an I/O operation. This will display an error message on the terminal that indicates the reason the operation failed.

WriteByte

One byte is written at the current file position which is then advanced by one byte.

WriteChar

A character is written at the current file position which is then advanced by one byte.

WriteRecord

A record is passed into this procedure as an ARRAY OF BYTE and written to a file in certain of bytes. This number usually is the size of the record in bytes. The actual number of bytes written is returned in a variable of type CARDINAL. It may be less than the requested number of bytes if there was an error.

WriteShortWord

A shortword is written at the current file position which is then advanced by one shortword (two bytes).

WriteWord

A word is is written at the current file position which is then advanced by one word.

InOut

There are seven procedures to facilitate general-purpose, high-level sequential I/O for file or terminal. Besides, the end of line constant is also defined.

EOL

This is an End-of-Line or linefeed character represented as a hexadecimal number "012".

Write

Print a character in the output stream.

WriteCard

There is a parameter specifying the number of character that the cardinal number is going to be written in. If the number is greater than the number of digits needed, then blanks are added in front of the number. If the number is less than the number of digits needed, then it is ignored.

WriteHex

Same as "WriteCard" except that a hexadecimal number is written.

WriteInt

Same as "WriteCard" except that an integer is written.

WriteLn

A linefeed is put in the output stream. It is equivalent to "Write(EOL)".

WriteOct

Same as "WriteCard" except that an octal number is written.

WriteString

The string specified in the parameter is written in the output stream.

Storage

ALLOCATE

This procedure allocates a block of memory of a designated size in bytes and returns the starting address of the allocated block. If the requested number of bytes are not available from the heap, then the passed-in address parameter will not be changed and an error message "Heap overflow" will return.

DEALLOCATE

A designated block of memory is freed in the execution of this procedure. This block of memory should be previously allocated with "ALLOCATE" and is made available for subsequent allocation but its contents is left undisturbed. An invalid of the block will cause the error message "Memory fault - core dumped".

Terminal

There are four procedures handling terminal I/O routines.

Read

"Read" returns the next character typed on the keyboard. This routine will block the execution of a program until a character is typed.

Write

A character is put on the terminal screen.

WriteLn

A lined character is sent to the terminal screen.

WriteString

The string specified in the parameter is printed on the terminal screen.

APPENDIX 2

Draft BSI Standard I/O Library of Modula-2

DEFINITION MODULE BoolIO;

FROM IO IMPORT Channel;

EXPORT QUALIFIED CanRead, CanSkip, Value, Skip,
Read, Write, Print;

```
(*
 * Boolean objects ignore all leading white
 * space (Space, Tab, LF, CR etc) and are
 * terminated by the first character that would
 * be illegal in the object.
 * This illegal character is left on the channel.
 *
 * If no data is currently available then the
 * predicates wait for some.
 *
 * CanRead returns TRUE if the current object
 * can be represented.
 * CanRead implies CanSkip.
 *
 * CanSkip returns TRUE if there is an object
 * available.
 * CanSkip does not imply CanRead as an object
 * could be well-formed but out of range.
 *
 * Value returns the current object.
 * CanRead must be TRUE.
 * If a call to CanRead currently returns FALSE
 * then Value will fail.
 *
 * Skip skips over the current object.
 * CanSkip must be TRUE.
 * if a call to CanSkip currently returns FALSE
 * then Skip will fail.
 *
 * Read places the current object into its
 * second parameter and then skips over it.
 * CanRead must TRUE.
 * If a call to CanRead currently returns FALSE
 * then Read will fail.
 *
 * Write the object to the channel with no padding.
 *)
```



```
DEFINITION MODULE Cardio;  
FROM IO IMPORT Channel;  
EXPORT QUALIFIED CanRead, CanSkip, Value, Skip,  
Read, Write, Print;  
  
(*  
 * Refer to the comments in BoolIO.  
 *)  
PROCEDURE CanRead (C: Channel) : BOOLEAN;  
PROCEDURE CanSkip (C: Channel) : BOOLEAN;  
PROCEDURE Value (C: Channel) : CARDINAL;  
PROCEDURE Skip (VAR C: Channel);  
PROCEDURE Read (VAR C: Channel;  
VAR Card: CARDINAL);  
PROCEDURE Write (VAR C: Channel;  
Card: CARDINAL);  
PROCEDURE Print (VAR C: Channel;  
Card: CARDINAL;  
Length: CARDINAL;  
RightJustified: BOOLEAN);  
END Cardio;
```

```
DEFINITION MODULE IntIO;
FROM IO IMPORT Channel;
EXPORT QUALIFIED CanRead, CanSkip, Value, Skip,
                Read, Write, Print;

(*
 * Refer to the comments in BoolIO.
 *)
PROCEDURE CanRead (C: Channel) : BOOLEAN;
PROCEDURE CanSkip (C: Channel) : BOOLEAN;
PROCEDURE Value (C: Channel) : INTEGER;
PROCEDURE Skip (VAR C: Channel);
PROCEDURE Read (VAR C: Channel;
                VAR Int: INTEGER);
PROCEDURE Write (VAR C: Channel;
                 Int: INTEGER);
PROCEDURE Print (VAR C: Channel;
                 Int: INTEGER;
                 Length: CARDINAL;
                 RightJustified: BOOLEAN);
END IntIO;
```

```
DEFINITION MODULE RealIO;
FROM IO IMPORT Channel;
EXPORT QUALIFIED CanRead, CanSkip, Value, Skip,
                Read, Write, Print;

(*
 * Refer to the comments in BoolIO.
 *)
PROCEDURE CanRead (C: Channel) : BOOLEAN;
PROCEDURE CanSkip (C: Channel) : BOOLEAN;
PROCEDURE Value (C: Channel) : REAL;
PROCEDURE Skip (VAR C: Channel);
PROCEDURE Read (VAR C: Channel;
                VAR Real: REAL);
PROCEDURE Write (VAR C: Channel;
                 Real: REAL);
PROCEDURE Print (VAR C: Channel;
                 Real: REAL;
                 Length: CARDINAL;
                 RightJustified: BOOLEAN);
END RealIO;
```

APPENDIX 3

BSI Modula-2 Utility Library Draft of 07 December 1984

```
DEFINITION MODULE SYSTEMToBeImpl;
(* This module is provide so that subswquent modules
   compile correctly, and will disappear as revised
   compilers become available. *)

FROM SYSTEM IMPORT WORD;

EXPORT QUALIFIED
    BYTE, ADDRESSINC, BUTECOUNT;

TYPE
    BYTE          = WORD;
    ADDRESSINC    = CARDINAL;
    BUTECOUNT     = CARDINAL;(* doubt aoubt this one *)

END SYSTEMToBeImpl.
```

DEFINITION MODULE Files;

EXPORT QUALIFIED

```
File,           FileState,
BinTextMode,   ReadWriteMode,  ReplaceMode,
Open,          Create,         Close,         Remove,
Reset,         Rewrite,        Truncate,     Flush,
EOF,           State,          ResetState,
GetFileName;
```

TYPE

```
File;
BinTextMode   = (BinMode, texMode);
ReadWriteMode = (readOnly, readWrite, appendOnly);
ReplaceMode   = (noReplace, replace);
FileState     = (ok,
                 nameError, noFile, existingFile,
                 deviceError, noMoreRoom, accessError,
                 notOpen, endError OutsideFile,
                 otherError);
```

```
PROCEDURE Open (VAR file      : File;
                name         : ARRAY OF CHAR;
                binText      : BinTextMode;
                writemode    : ReadWriteMode;
                VAR state    : FileState);
```

```
PROCEDURE Create ( VAR file      : File;
                  name         : ARRAY OF CHAR;
                  binText      : BinTextMode;
                  repMode     : ReplaceMode;
                  VAR state    : FileState);
```

```
PROCEDURE Close (VAR file      : File;
                 VAR state    : FileState);
```

```
PROCEDURE Remove (VAR file      : File;
                  VAR state    : FileState);
```

```
PROCEDURE Reset (  file      : File;
                  VAR state  : FileState);
```

```
PROCEDURE Rewrite (  file      : File;
                   VAR state  : FileState);
```

```
PROCEDURE Truncate (  file      : File;
                    VAR state  : FileState);
```

```
PROCEDURE Flush (  file      : File;
```

```
VAR state      : FileState);  
PROCEDURE EOF (file : File)   : BOOLEAN;  
PROCEDURE State (file : File) : FileState;  
PROCEDURE ResetState ( file : File;  
                      VAR state : FileState);  
PROCEDURE GetFileName ( file : File;  
                      VAR name : ARRAY OF CHAR;  
                      VAR state : FileState);  
  
END Files.
```



```
DEFINITION MODULE Text;
FROM Files IMPORT File, FileState;
EXPORT QUALIFIED
    EOL,
    ReadChar, ReadLn, ReadString,
    UndoRead, CondRead,
    WriteChar, WriteString, WriteLn;
PROCEDURE EOL (file : File) : BOOLEAN;
PROCEDURE ReadChar ( file : File;
                    VAR ch : CHAR;
                    VAR state : FileState);
PROCEDURE ReadString ( file : File;
                    VAR str : ARRAY OF CHAR;
                    VAR state : FileState);
PROCEDURE ReadLn ( file : File;
                 VAR state : FileState);
PROCEDURE UndoRead ( file : File;
                   VAR state : FileState);
PROCEDURE CondRead ( file : File;
                   VAR ch : CHAR;
                   VAR success : BOOLEAN;
                   VAR state : FileState);
PROCEDURE WriteChar ( file : File;
                    ch : CHAR;
                    VAR state : FileState);
PROCEDURE WriteString ( file : File;
                    str : ARRAY OF CHAR;
                    VAR state : FileState);
PROCEDURE WriteLn ( file : File;
                  VAR state : FileState);
END Text.
```

```
DEFINITION MODULE NumberIO;

FROM Files      IMPORT File, FileState;
FROM SYSTEM     IMPORT WORD;

EXPORT QUALIFIED
  ReadInt,      ReadCard,      ReadNum,
  WriteInt,     WriteCard,     WriteNum;

PROCEDURE ReadInt (   file      : File;
                    VAR int     : INTEGER;
                    VAR success : BOOLEAN;
                    VAR state   : FileState);

PROCEDURE ReadCard (  file      : File;
                    VAR card    : CARDINAL;
                    VAR success : BOOLEAN;
                    VAR state   : FileState);

PROCEDURE ReadNum (  file      : File;
                    VAR num     : WORD;
                    base       : CARDINAL;
                    VAR success : BOOLEAN;
                    VAR state   : FileState);

PROCEDURE WriteInt(  file      : File;
                    int       : INTEGER;
                    width     : CARDINAL;
                    VAR state  : FileState);

PROCEDURE WriteCard (  file      : File;
                    card      : CARDINAL;
                    width     : CARDINAL;
                    VAR state   : FileState);

PROCEDURE WriteNum (  file      : File;
                    num       : WORD;
                    base       : CARDINAL;
                    width     : CARDINAL;
                    VAR state   : FileState);

END NumberIO.
```

```
DEFINITION MODULE FilePositions;

FROM Files          IMPORT File, FileState;
FROM SYSTEMToImpl  IMPORT ADDRESSINC;

EXPORT QUALIFIED
    FilePosition,
    GetFilePos,      SetFilePos,
    CalcFilePos,     GetEOF,      GetBOF;

TYPE
    FilePosition = RECORD END;

PROCEDURE GetFilePos (   file : File;
                       VAR pos : FilePosition);

PROCEDURE GetEOF (   file : File;
                   VAR pos : FilePosition);

PROCEDURE GetBOF (   file : File;
                   VAR pos : FilePosition);

PROCEDURE CalcFilePos ( file : File;
                       VAR pos : FilePosition;
                       numElements : INTEGER;
                       elementlength : ADDRESSINC);

PROCEDURE SetFilePos (   file : File;
                       pos : FilePosition;
                       VAR state : FileState);

END FilePosition.
```

```
DEFINITION MODULE Directory;
FROM Files      IMPORT FileState;
EXPORT QUALIFIED
  FileNameType,
  Rename,           Delete,
  DirQueryProc,    DirQuery,
  TypeOfFileName;

TYPE
  FileNameType = (invalidName, singleName, wildName);
  DirQueryProc = PROCEDURE (ARRAY OF CHAR, VAR BOOLEAN);

PROCEDURE Rename (   fromName  : ARRAY OF CHAR;
                   toName    : ARRAY OF CHAR;
                   VAR state  : FileState);

PROCEDURE Delete (  fileName  : ARRAY OF CHAR;
                   VAR state  : FileState);

PROCEDURE TypeOfFileName (name  : ARRAY OF CHAR)
                   : FileNameType;

PROCEDURE DirQuery ( wildName  : ARRAY OF CHAR;
                   dirProc   : DirQueryProc;
                   VAR state  : FileState);

END Directory.
```

```
DEFINITION MODULE SimpleIO;
FROM SYSTEM      IMPORT WORD;
EXPORT QUALIFIED
    EOT,          EOL,          ReadLn,          ReadInt,
    ReadChar,    ReadString,    CondRead,    UndoRead,
    ReadCard,    ReadNum,       WriteLn,     WriteInt,
    WriteChar,   WriteString,   WriteCard,   WriteNum;

PROCEDURE EOT ()           : BOOLEAN;
PROCEDURE EOL ()           : BOOLEAN;
PROCEDURE ReadChar (VAR ch : CHAR);
PROCEDURE ReadString (VAR str : ARRAY OF CHAR);
PROCEDURE ReadLn;
PROCEDURE ReadInt (VAR int : INTEGER;
                  VAR success : BOOLEAN);
PROCEDURE ReadCard (VAR card : CARDINAL;
                   VAR success : BOOLEAN);
PROCEDURE ReadNum (VAR num : WORD;
                  base : CARDINAL;
                  VAR success : BOOLEAN);
PROCEDURE CondRead (VAR ch : CHAR;
                   VAR success : BOOLEAN);
PROCEDURE UndoRead ();
PROCEDURE WriteChar ( ch : CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString ( str : ARRAY OF CHAR);
PROCEDURE WriteInt ( int : INTEGER;
                   width : CARDINAL);
PROCEDURE WriteCard ( card : CARDINAL;
                   width : CARDINAL);
PROCEDURE WriteNum ( num : WORD;
```

```
base      : CARDINAL;  
width     : CARDINAL);
```

END SimpleIO.

DEFINITION MODULE RealIO;

EXPORT QUALIFIED
 ReadReal, WriteReal;

```
PROCEDURE ReadReal (VAR real      : REAL;  
                   VAR success   : BOOLEAN);
```

```
PROCEDURE WriteReal ( real      : REAL;  
                    width     : CARDINAL;  
                    decPlaces  : INTEGER);
```

END RealIO.

```
DEFINITION MODULE StandardIO;
FROM Files      IMPORT File;
EXPORT QUALIFIED
    SetInput,   GetInput,       SetOutput,   GetOutput,
    EchoMode,  SetEchoMode,    GetEchoMode,
    GetErrorInput,
    LogMode,   SetLogMode,     GetErrorOutput,
    SetLog,    GetLog;         GetLogMode,

PROCEDURE Setinput ( input      : File);
PROCEDURE GetInput (VAR input   : File);
PROCEDURE SetOutput ( output    : File);
PROCEDURE GetOutput (VAR output : File);

TYPE
    EchoMode = (echo, noEcho);

PROCEDURE SetEchomode ( mode    : EchoMode);
PROCEDURE GetEchoMode ()      : EchoMode;
PROCEDURE GetErrorOutput (VAR errorFile : File);
PROCEDURE GetErrorInput (VAR errorFile : File);

TYPE
    LogMode = (loggingOn, loggingOff);

PROCEDURE SetLog ( log          : File);
PROCEDURE GetLog (VAR log       : File);
PROCEDURE SetLogMode ( mode     : LogMode);
PROCEDURE GetLogMode ()        : LogMode;

END StandardIO.
```

```
DEFINITION MODULE Terminal;

EXPORT QUALIFIED
    ReadChar,   ReadString,   CondRead,
    WriteChar, WriteString,   WriteLn;
    NumRows,   NumCols,      GotoRowCol,
    EraseScreen, EraseToEOL, EraseToEOS;

PROCEDURE ReadChar (VAR ch      : CHAR);

PROCEDURE ReadString (VAR str   : ARRAY OF CHAR);

PROCEDURE CondRead (VAR ch      : CHAR;
                   VAR success : BOOLEAN);

PROCEDURE WriteChar ( ch      : CHAR);

PROCEDURE WriteString ( str   : ARRAY OF CHAR);

PROCEDURE WriteLn;

PROCEDURE NumRows ()          : CARDINAL;

PROCEDURE NumCols ()         : CARDINAL;

PROCEDURE GotoRowCol ( row    : CARDINAL;
                     col     : CARDINAL);

PROCEDURE EraseScreen;

PROCEDURE EraseToEOL;

PROCEDURE EraseToEOS;

END Terminal.
```



```
DEFINITION MODULE MathLib;

EXPORT QUALIFIED
  Sqrt,      Exp,   Ln,   Sin,
  Cos,      Arctan, Entier, Power;

PROCEDURE Sqrt ( real : REAL ) : REAL;
PROCEDURE Exp  ( real : REAL ) : REAL;
PROCEDURE Ln   ( real : REAL ) : REAL;
PROCEDURE Sin  ( real : REAL ) : REAL;
PROCEDURE Cos  ( real : REAL ) : REAL;
PROCEDURE Arctan( real : REAL ) : REAL;
PROCEDURE Entier( real : REAL ) : INTEGER;
PROCEDURE Power ( real : REAL
                  exp  : REAL ) : REAL;

END MathLib.
```



```
DEFINITION MODULE String;

EXPORT QUALIFIED
    CompareResult,
    Length,      Assign,      Insert,      Delete,
    Position,    Substring,    Compare,     Concat;

TYPE
    CompareResult = (less, equal, greater);

PROCEDURE Length ( str          : ARRAY OF CHAR)
                  : CARDINAL;

PROCEDURE Assign (   source     : ARRAY OF CHAR;
                   VAR dest     : ARRAY OF CHAR;
                   VAR success   : BOOLEAN);

PROCEDURE Insert (   source     : ARRAY OF CHAR;
                   VAR dest     : ARRAY OF CHAR;
                   index        : CARDINAL;
                   VAR success   : BOOLEAN);

PROCEDURE Delete (VAR str       : ARRAY OF CHAR;
                  index        : CARDINAL;
                  len          : CARDINAL;
                  VAR success   : BOOLEAN);

PROCEDURE Position (pattern     : ARRAY OF CHAR;
                   source       : ARRAY OF CHAR;
                   VAR index     : CARDINAL;
                   VAR found     : BOOLEAN);

PROCEDURE Substring ( source     : ARRAY OF CHAR;
                    index        : CARDINAL;
                    len          : CARDINAL;
                    VAR dest     : ARRAY OF CHAR;
                    VAR success   : BOOLEAN);

PROCEDURE Concat (   source1    : ARRAY OF CHAR;
                   source2     : ARRAY OF CHAR;
                   VAR dest     : ARRAY OF CHAR;
                   VAR success   : BOOLEAN);

PROCEDURE Compare ( string1     : ARRAY OF CHAR;
                  string2      : ARRAY OF CHAR)
                  : CompareResult;

END String.
```

DEFINITION MODULE Convert;

EXPORT QUALIFIED

IntToStr,	StrToInt,
CardToStr,	StrToCard,
NumToStr,	StrToNum;

PROCEDURE IntToStr (int : INTEGER;
 VAR str : ARRAY OF CHAR;
 width : CARDINAL;
 VAR success : BOOLEAN);

PROCEDURE CardToStr (card : CARDINAL;
 VAR str : ARRAY OF CHAR;
 width : CARDINAL;
 VAR success : BOOLEAN);

PROCEDURE NumToStr (num : CARDINAL;
 VAR str : ARRAY OF CHAR;
 base : CARDINAL;
 width : CARDINAL;
 VAR success : BOOLEAN);

PROCEDURE StrToInt (str : ARRAY OF CHAR;
 VAR int : INTEGER;
 VAR success : BOOLEAN);

PROCEDURE StrToCard (str : ARRAY OF CHAR;
 VAR card : CARDINAL;
 VAR success : BOOLEAN);

PROCEDURE StrToNum (str : ARRAY OF CHAR;
 VAR num : CARDINAL;
 base : CARDINAL;
 VAR success : BOOLEAN);

END Convert.

```
DEFINITION MODULE ConvertReal;
EXPORT QUALIFIED
    RealToStr, StrToReal;

PROCEDURE RealToStr (    real    : REAL;
                       VAR str   : ARRAY OF CHAR;
                       width    : CARDINAL;
                       decPlaces : INTEGER;
                       VAR success: BOOLEAN);

PROCEDURE StrToReal (    str     : ARRAY OF CHAR;
                       VAR real  : REAL;
                       VAR success: BOOLEAN);

END ConvertReal.
```

APPENDIX 4

Programs Listing

```
DEFINITION MODULE Arguments;

  FROM SYSTEM IMPORT ADDRESS;

  VAR
    ArgCount : CARDINAL;

  (* ArgCount represents the number of arguments from 0
     to ArgCount - 1. *)
  (* Argument 0 is always the filename *)

  PROCEDURE GetArgument (argnum : CARDINAL;
                        VAR arg : ARRAY OF CHAR;
                        VAR length : CARDINAL);

  PROCEDURE GetEnvironment (name : ARRAY OF CHAR;
                           VAR value : ARRAY OF CHAR;
                           VAR OK : BOOLEAN);

  PROCEDURE PutEnvironment (string : ARRAY OF CHAR;
                            VAR OK : BOOLEAN);

  PROCEDURE GetOpt(optstr : ARRAY OF CHAR;
                  VAR argstr : ARRAY OF CHAR;
                  VAR optind : LONGINT) : CHAR;

END Arguments.
```

DEFINITION MODULE Clock;

FROM CLibrary CIMPORT TimeType, tm;

(* TimeType = LONGCARD; *)

(* TimeType is already imported from CLibrary,

* there is no need to define it again in Clock.

* tm = RECORD

tm_sec : TimeType;	second, 0 - 59
tm_min : TimeType;	minutes, 0 - 59
tm_hour : TimeType;	hours, 0 - 23
tm_mday : TimeType;	day of month, 1 - 31
tm_mon : TimeType;	month of year, 1 - 12
tm_year : TimeType;	year - 1900
tm_wday : TimeType;	day of week, Sunday = 0
tm_yday : TimeType;	day of year, 0 - 365
tm_isdst : TimeType;	> 0 = Daylight savings time is in effect

END; *)

TYPE

TimeRecType = tm;

PROCEDURE Time(VAR timerec : TimeRecType);

PROCEDURE ClockTime (VAR time: TimeType);

(* returns current time as # seconds *)

PROCEDURE Date (VAR date: ARRAY OF CHAR);

(* returns current time as string, min length 26 chars *)

PROCEDURE ConvertTime (time: TimeType;

VAR date: ARRAY OF CHAR);

(* converts time given as argument into string,
min length 26 chars *)

END Clock.

```
DEFINITION MODULE Coroutine;  
FROM SYSTEM IMPORT ADDRESS;  
PROCEDURE NEWPROCESS(p: PROC; wspaceadr: ADDRESS;  
                    wspacesize: LONGCARD; VAR cor: ADDRESS);  
PROCEDURE TRANSFER(VAR source, destination : ADDRESS);  
END Coroutine.
```



```
DEFINITION MODULE Conversions;

FROM SYSTEM IMPORT BYTE, WORD;

PROCEDURE Done(): BOOLEAN;
(* returns TRUE if last String/Number conversion
   was successful *)
(* number to string conversions *)

PROCEDURE NumToString (num: WORD;
                       len, base: CARDINAL;
                       VAR s: ARRAY OF CHAR);

PROCEDURE OctToString (num: WORD;
                       len: CARDINAL;
                       VAR s: ARRAY OF CHAR);

PROCEDURE HexToString (num: WORD;
                       len: CARDINAL;
                       VAR s: ARRAY OF CHAR);

PROCEDURE IntToString (num: INTEGER;
                       len: CARDINAL;
                       VAR s: ARRAY OF CHAR);

PROCEDURE CardToString (num: CARDINAL;
                        len: CARDINAL;
                        VAR s: ARRAY OF CHAR);

PROCEDURE RealToString (num : REAL;
                        len : CARDINAL;
                        VAR s : ARRAY OF CHAR);

(* string to number conversions *)

PROCEDURE StringToOct (s: ARRAY OF CHAR): CARDINAL;

PROCEDURE StringToHex (s: ARRAY OF CHAR): CARDINAL;

PROCEDURE StringToInt (s: ARRAY OF CHAR): INTEGER;
```

```
PROCEDURE StringToCard (s: ARRAY OF CHAR): CARDINAL;  
PROCEDURE StringToReal (s : ARRAY OF CHAR) : REAL;  
(* number to number conversions *)  
PROCEDURE ByteToInt (num: BYTE): INTEGER;  
END Conversions.
```

DEFINITION MODULE CLibrary;

FROM SYSTEM IMPORT ADDRESS, BYTE, WORD;

CONST

```
EOF = -1;
NULL = 0;
NFILE = 20;
IOREAD = 0; (*file open for reading *)
IOWRT = 1; (*file open for writing *)
IONBF = 2; (*file is unbuffered *)
IOMYBUF = 3; (*big buffer allocated *)
IOEOF = 4; (*EOF has occurred on this file *)
IOERR = 5; (*error has occurred on this file *)
IOUNK = 6; (*indicates buffering status unknown*)
IORW = 7;
```

(* Allowed values of errno *)

```
EOK = 0; (* No error *)
EPERM = 1; (* Not super-user *)
ENOENT = 2; (* No such file or directory *)
ESRCH = 3; (* No such process *)
EINTR = 4; (* interrupted system call *)
EIO = 5; (* I/O error *)
ENXIO = 6; (* No such device or address *)
E2BIG = 7; (* Arg list too long *)
ENOEXEC = 8; (* Exec format error *)
EBADF = 9; (* Bad file number *)
ECHILD = 10; (* No children *)
EAGAIN = 11; (* No more processes *)
ENOMEM = 12; (* Not enough core *)
EACCES = 13; (* Permission denied *)
EFAULT = 14; (* Bad address *)
ENOTBLK = 15; (* Block device required *)
EBUSY = 16; (* Mount device busy *)
EEXIST = 17; (* File exists *)
EXDEV = 18; (* Cross-device link *)
ENODEV = 19; (* No such device *)
ENOTDIR = 20; (* Not a directory *)
EISDIR = 21; (* Is a directory *)
EINVAL = 22; (* Invalid argument *)
ENFILE = 23; (* File table overflow *)
EMFILE = 24; (* Too many open files *)
ENOTTY = 25; (* Not a typewriter *)
ETXTBSY = 26; (* Text file busy *)
EFBIG = 27; (* File too large *)
ENOSPC = 28; (* No space left on device *)
ESPIPE = 29; (* Illegal seek *)
```

```
EROFS      = 30; (* Read only file system      *)
EMLINK     = 31; (* Too many links             *)
EPIPE     = 32; (* Broken pipe                 *)
EDOM      = 33; (* Math arg out of domain of func *)
ERANGE    = 34; (* Math result not representable *)
ENOMSG    = 35; (* No message of desired type      *)
EIDRM     = 36; (* IPC identifier removed         *)
EDEADLK   = 45; (* File region locking deadlock  *)
```

(* Constants for access functions *)

```
R_OK = 2;
W_OK = 1;
X_OK = 0; (* bit on one *)
F_OK = 0; (* bit on zero *)
```

(* Constants for fseek *)

```
L_SET = 0;
L_CUR = 1;
L_END = 2;
```

NAME_MAX = 14; (* Max no. of characters in a file name *)

TYPE

```
ProcessIDType = LONGINT;
InoType = WORD;
DevType = WORD;
OffType = LONGINT;
TimeType = LONGCARD;
FileRange = [0..NFILE - 1];
timestring = ARRAY [0..25] OF CHAR;
```

FileBlock =

```
RECORD
  ptr : ADDRESS; (* next character position *)
  cnt : LONGINT; (* number of characters left *)
  base : ADDRESS; (* location of buffer *)
  flag : BYTE; (* mode of file access *)
  fd : BYTE (* file descriptor *)
END;
```

IOBlock = ARRAY FileRange OF FileBlock;

```
SignalType = (SigReserved, (* may not be used *)
              SigHup,      (* hangup *)
              SigInt,      (* interrupt *)
              SigQuit,     (* quit *)
              SigIll,      (* illegal instruction *)
              SigTrap,     (* trace trap *)
```

```
SigIot,      (* IOT instruction      *)
SigEmt,      (* EMT instruction      *)
SigFpe,      (* floating point exception*)
SigKill,     (* kill                  *)
SigBus,      (* bus error            *)
SigSegv,     (* segmentation violation *)
SigSys,      (* bad argument to system
              call *)
SigPipe,     (* write on a pipe with no
              one to read it *)
SigAlrm,     (* alarm clock          *)
SigTerm,     (* software termination
              signal from kill *)
SigUsr1,     (* user defined signal 1 *)
SigUsr2,     (* user defined signal 2 *)
SigChld,     (* death of a child     *)
SigPwr,      (* power fail           *) );
```

```
SignalProc = PROCEDURE (SignalType);
```

```
tms = RECORD
  tms_utime : TimeType;
  (* The CPU time used while executing instructions
  in the user space of the calling process *)
  tms_stime : TimeType;
  (* The CPU time used by the system on behalf of
  the calling process *)
  tms_cutime: TimeType;
  (* The sum of the tms_utimes and tms_cutimes
  of the child process *)
  tms_cstime : TimeType;
  (* The sum of the tms_stimes and tms_cstimes
  of the child process *)
END;
```

```
tm = RECORD
  tm_sec : TimeType; (* second, 0 - 59 *)
  tm_min : TimeType; (* minutes, 0 - 59 *)
  tm_hour : TimeType; (* hours, 0 - 23 *)
  tm_mday : TimeType; (* day of month, 1 - 31 *)
  tm_mon : TimeType; (* month of year, 1 - 12 *)
  tm_year : TimeType; (* year - 1900 *)
  tm_wday : TimeType; (* day of week, Sunday = 0 *)
  tm_yday : TimeType; (* day of year, 0 - 365 *)
  tm_isdst : TimeType; (* > 0 = Daylight savings
                        time is in effect *)
END;
```

```
statype = RECORD
  st_mode : WORD; (* File node *)
```

```
st_ino      : InoType; (* Inode number *)
st_dev      : DevType;
(* ID of device containing a directory
entry for this file *)
st_rdev     : DevType;
(* ID of device. This entry is defined
only for character special or block
special files *)
st_nlink    : WORD; (* Number of links *)
st_uid     : WORD;
(* User ID of the file's owner *)
st_gid     : WORD;
(* Group ID of the file's group *)
st_size    : OffType;
(* File size in bytes *)
st_atime   : TimeType;
(* Time of last access *)
st_mtime   : TimeType;
(* Time of last data modification *)
st_ctime   : TimeType
(* Time of last file status change *)
END;
```

```
intPtr = POINTER TO LONGINT;
charPtr = POINTER TO CHAR;
Stream = POINTER TO FileBlock;
tmsPtr = POINTER TO tms;
tmPtr = POINTER TO tm;
timestringPtr = POINTER TO timestring;
statPtr = POINTER TO stattype;
```

VAR

```
_iob : IOBlock; (* extern _iob in file.h *)
errno : LONGINT; (* extern errno in errno.h *)
optarg : ADDRESS; (* return the option argument *)
optind : LONGINT; (* return the option index *)
```

PROCEDURE abort () : LONGINT;

PROCEDURE access (path : ARRAY OF CHAR ;
mode : BITSET) : LONGINT;

PROCEDURE chdir (path : ARRAY OF CHAR ;
mode : BITSET) : LONGINT;

PROCEDURE chmod (path : ARRAY OF CHAR ;
mode : BITSET) : LONGINT;

PROCEDURE chown (path : ARRAY OF CHAR ;
owner, group : BITSET) : LONGINT;

```
PROCEDURE close ( fileds : LONGINT ) : LONGINT;
PROCEDURE creat ( filename : ARRAY OF CHAR ;
                 mode : BITSET ) : LONGINT;
PROCEDURE dup ( fileds : LONGINT ) : LONGINT;
PROCEDURE exit ( status : LONGINT );
PROCEDURE fclose ( stream : Stream );
PROCEDURE fflush ( stream : Stream ) : LONGINT;
PROCEDURE fopen ( filename : ARRAY OF CHAR;
                 type : ARRAY OF CHAR ) : Stream;
PROCEDURE freopen ( filename, type : ARRAY OF CHAR;
                  stream : Stream ) : Stream;
PROCEDURE fdopen ( fileds : LONGINT ;
                  type : ARRAY OF CHAR ) : Stream;
PROCEDURE fread ( ptr : ARRAY OF BYTE ;
                 size, nitems : LONGINT ;
                 stream : Stream ) : LONGINT;
PROCEDURE fwrite ( ptr : ARRAY OF BYTE ;
                  size, nitems : LONGINT ;
                  stream : Stream ) : LONGINT;
PROCEDURE fseek ( stream : Stream ;
                 offset : LONGINT ;
                 ptrname : LONGINT ) : LONGINT;
PROCEDURE rewind ( stream : Stream ) : LONGINT;
PROCEDURE ftell ( stream : Stream ) : LONGINT;
PROCEDURE getcwd ( buf : ARRAY OF CHAR ;
                  size : LONGINT ) : ADDRESS;
PROCEDURE link ( filename1, filename2 : ARRAY OF CHAR )
               : LONGINT;
PROCEDURE malloc ( size : LONGCARD ) : ADDRESS;
PROCEDURE free ( ptr : ADDRESS );
PROCEDURE realloc ( ptr : ADDRESS ;
                  size : LONGCARD ) : ADDRESS;
```

```
PROCEDURE mknod ( VAR path : ARRAY OF CHAR ;
                  mode : BITSET ;
                  dev : LONGINT ) : LONGINT;

PROCEDURE system ( VAR string : ARRAY OF CHAR ) : LONGINT;

PROCEDURE unlink ( VAR filename : ARRAY OF CHAR ) : LONGINT;

PROCEDURE setbuf ( stream : Stream;
                  VAR buf : ARRAY OF CHAR ) : LONGINT;
    (* IO Functions *)

PROCEDURE fgetc ( stream : Stream ) : LONGINT;

PROCEDURE fputc ( c : LONGINT;
                  stream : Stream ) : LONGINT;

PROCEDURE fputs ( VAR str : ARRAY OF CHAR ;
                  stream : Stream ) : LONGINT;

PROCEDURE sscanf ( VAR s : ARRAY OF CHAR;
                   VAR format : ARRAY OF CHAR;
                   r: ADDRESS) : LONGINT;

PROCEDURE sprintf ( VAR s : ARRAY OF CHAR;
                   VAR format : ARRAY OF CHAR;
                   HighReal : LONGINT;
                   LowReal : LONGINT) : LONGINT;

PROCEDURE getenv ( VAR name : ARRAY OF CHAR ) : ADDRESS;

PROCEDURE putenv ( VAR string : ARRAY OF CHAR ) : LONGINT;
    (* time and date *)

PROCEDURE times ( buffer : tmsPtr ) : LONGINT;
    (* Get process and child process times *)

PROCEDURE time ( tloc : intPtr) : LONGINT; (* Get time *)

PROCEDURE stime ( tp : intPtr) : LONGINT;

PROCEDURE ctime ( clock : intPtr) : timestringPtr;

PROCEDURE localtime ( clock : intPtr) : tmPtr;

PROCEDURE gmtime ( clock : intPtr) : tmPtr;

PROCEDURE asctime ( tm : tmsPtr) : timestringPtr;

(* get file status *)
```



```
PROCEDURE stat ( VAR path : ARRAY OF CHAR;  
                buf : statPtr) : LONGINT;  
  
PROCEDURE fstat ( fileds : LONGINT;  
                buf : statPtr) : LONGINT;  
  
(* temporary file creation *)  
  
PROCEDURE mktemp ( VAR template : ARRAY OF CHAR) : charPtr;  
  
PROCEDURE tmpfile () : Stream;  
  
PROCEDURE tmpnam ( VAR s : ARRAY OF CHAR) : charPtr;  
  
PROCEDURE tempnam ( VAR dir : ARRAY OF CHAR;  
                  VAR pfx : ARRAY OF CHAR) : charPtr;  
  
PROCEDURE getpid () : ProcessIDType;  
  
PROCEDURE getpgrp () : ProcessIDType;  
  
PROCEDURE getppid () : ProcessIDType;  
  
PROCEDURE kill ( pid : ProcessIDType;  
               sig : SignalType) : LONGINT;  
  
PROCEDURE fork () : ProcessIDType;  
  
PROCEDURE pause () : LONGINT;  
  
PROCEDURE sleep ( seconds : CARDINAL) : CARDINAL;  
  
PROCEDURE signal ( Sig : SignalType;  
                 Func : SignalProc);  
  
PROCEDURE wait ( stat_loc : intPtr ) : LONGINT;  
  
PROCEDURE alarm ( sec : CARDINAL) : CARDINAL;  
  
PROCEDURE gsignal ( sig : SignalType) : LONGINT;  
  
PROCEDURE ssignal ( Sig : SignalType;  
                  action : SignalProc);  
  
PROCEDURE execv ( VAR path : ARRAY OF CHAR;  
                 argv : ADDRESS) : LONGINT;  
  
PROCEDURE rand () : CARDINAL;  
  
PROCEDURE srand ( seed : CARDINAL);
```

```
PROCEDURE getopt(argc : LONGCARD; argv : ADDRESS;
                 optstring : ARRAY OF CHAR) : LONGINT;
(* routines for 32 bit integer and 32 & 64 bit
float arithmetic *)

PROCEDURE dbtofl__ ;
(* convert LONGREAL to REAL *)

PROCEDURE fltodb__ ;
(* convert REAL to LONGREAL *)

PROCEDURE ltodb__ ;
(* convert LONGINT to LONGREAL *)

PROCEDURE dbtol__ ;
(* convert LONGREAL to LONGINT *)

PROCEDURE dbadd__(addend: LONGREAL);
(* add 2 LONGREALs *)

PROCEDURE dbsub__(subtrahend: LONGREAL);
(* subtract 2 LONGREALs *)

PROCEDURE dbmul__(multiplier: LONGREAL);
(* multiply 2 LONGREALs *)

PROCEDURE dbdiv__(divisor: LONGREAL);
(* divide 2 LONGREALs *)

END CLibrary.

DEFINITION MODULE ErrorHandling;

PROCEDURE HaltMessage (Str : ARRAY OF CHAR);
(* writes the str on Stderr and halt the program *)

PROCEDURE ErrorMessage (Str : ARRAY OF CHAR);
(* writes the str on Stderr and continue the program *)

END ErrorHandling.
```

DEFINITION MODULE FileInformation;

FROM CLibrary CIMPORT stat, fstat, InoType, DevType,
OffType, TimeType;

FROM SYSTEM IMPORT WORD;

(* There are two sets of procedures for operating
* system programming. They contains the same
* functionalities, but one set takes the path
* name as the input and the other file descriptor. *)

PROCEDURE GetMode (path : ARRAY OF CHAR) : WORD;
(* File mode *)

PROCEDURE GetIno (path : ARRAY OF CHAR) : InoType;
(* Inode number *)

PROCEDURE GetDev (path : ARRAY OF CHAR) : DevType;
(* ID of device containing a directory entry for this file *)

PROCEDURE GetRdev (path : ARRAY OF CHAR) : DevType;
(* ID of device. This entry is defined only for character
special or block special file *)

PROCEDURE GetNlink (path : ARRAY OF CHAR) : WORD;
(* Number of links *)

PROCEDURE GetUid (path : ARRAY OF CHAR) : WORD;
(* User ID of the file's owner *)

PROCEDURE GetGid (path : ARRAY OF CHAR) : WORD;
(* Group ID of the file's group *)

PROCEDURE GetSize (path : ARRAY OF CHAR) : OffType;
(* File size in bytes *)

PROCEDURE GetAtime (path : ARRAY OF CHAR) : TimeType;
(* Time of last access *)

PROCEDURE GetMtime (path : ARRAY OF CHAR) : TimeType;
(* Time of last data modification *)

PROCEDURE GetCtime (path : ARRAY OF CHAR) : TimeType;
(* Time of last file status change *)

PROCEDURE fGetMode (filedescr: LONGINT) : WORD;

```
(* File mode *)  
  
PROCEDURE fGetIno (filedescr: LONGINT) : InoType;  
(* Inode number *)  
  
PROCEDURE fGetDev (filedescr: LONGINT) : DevType;  
(* ID of device containing a directory entry for this file *)  
  
PROCEDURE fGetRdev (filedescr: LONGINT) : DevType;  
(* ID of device. This entry is defined only for character  
special or block special file *)  
  
PROCEDURE fGetNlink (filedescr: LONGINT) : WORD;  
(* Number of links *)  
  
PROCEDURE fGetUid (filedescr: LONGINT) : WORD;  
(* User ID of the file's owner *)  
  
PROCEDURE fGetGid (filedescr: LONGINT) : WORD;  
(* Group ID of the file's group *)  
  
PROCEDURE fGetSize (filedescr: LONGINT) : OffType;  
(* File size in bytes *)  
  
PROCEDURE fGetAtime (filedescr: LONGINT) : TimeType;  
(* Time of last access *)  
  
PROCEDURE fGetMtime (filedescr: LONGINT) : TimeType;  
(* Time of last data modification *)  
  
PROCEDURE fGetCtime (filedescr: LONGINT) : TimeType;  
(* Time of last file status change *)  
  
END FileInformation.
```

```
DEFINITION MODULE FileSystem;

FROM SYSTEM IMPORT ADDRESS, BYTE, WORD;

FROM CLibrary CIMPORT NAME_MAX, Stream;

CONST
  EOL = 12C; (* <LF> *) (*U3B1*)
  MAX_FILENAME = 127;

TYPE
  File = Stream;
  FileName = ARRAY [0..MAX_FILENAME] OF CHAR;
  ModeType = (O_RDONLY, O_WRONLY, O_APPEND,
              O_RDWR, O_WRONLY, O_APPENDUPDATE);

VAR
  Stdin, Stdout, Stderr : File;
  Base_iob : ADDRESS;

PROCEDURE Done(): BOOLEAN;
(* should be called after each FileSystem procedure
   to determine if previous call was successful. *)

PROCEDURE Create(VAR f: File;
                 filename: ARRAY OF CHAR);
(* create a new file and open it for read and
   write access. *)

PROCEDURE CreateTempfile (VAR f : File;
                          filename : ARRAY OF CHAR);
(* create a new file and open it for read and
   write access. The parameter filename should look
   like a file name with six trailing Xs *)

PROCEDURE Open(VAR f: File;
               filename: ARRAY OF CHAR;
               mode : ModeType);
(* opens an existing file.
   mode see ModeType above
   position := beginning of file *)

PROCEDURE Close (VAR f: File);
(* closes file, modifications become permanent *)

PROCEDURE Release(VAR f: File);
(* if write access was allowed on the file, it is
   deleted otherwise it is closed *)
```

```
PROCEDURE Rename(old,new: ARRAY OF CHAR);
PROCEDURE Delete(filename: ARRAY OF CHAR);
PROCEDURE GetPos(VAR f: File;
                 VAR position: LONGCARD);
(* return position *)
PROCEDURE SetPos(VAR f: File;
                 position: LONGCARD);
(* set position to position which is the byte
   position within the file. *)
PROCEDURE Reset(VAR f: File);
(* position := beginning of file *)
PROCEDURE ReadMode(VAR f : File) : BOOLEAN;
PROCEDURE WriteMode(VAR f : File) : BOOLEAN;
PROCEDURE FileNotBuffered(VAR f : File) : BOOLEAN;
PROCEDURE FileBigBuffer(VAR f : File) : BOOLEAN;
PROCEDURE Eof(VAR f: File): BOOLEAN;
(* position = end of file *)
PROCEDURE ErrorOnFile(VAR f : File) : BOOLEAN;
PROCEDURE ReadWriteMode(VAR f : File) : BOOLEAN;
(***** binary access *****)
PROCEDURE ReadWord (VAR f: File;
                   VAR w: WORD);
(* read one word from position; advance position *)
PROCEDURE ReadByte (VAR f: File;
                   VAR b: BYTE);
(* read one byte from position; advance position *)
PROCEDURE WriteWord (VAR f: File;
                    w: WORD);
(* write one word at position; advance position *)
PROCEDURE WriteByte (VAR f: File;
                    b: BYTE);
(* write one byte at position; advance position *)
PROCEDURE Flush (VAR f : File);
```

```
(***** textfile access *****)  
  
PROCEDURE ReadChar (VAR f: File;  
                   VAR ch: CHAR);  
(*write one character from position; advance position*)  
  
PROCEDURE WriteChar(VAR f: File;  
                   ch: CHAR);  
(* write one character at position; advance position *)  
  
PROCEDURE WriteString (VAR f : File;  
                      str : ARRAY OF CHAR);  
(* write a string from position;  
  advance StringLen(str) byte position on the file *)  
  
PROCEDURE ReadRecord (VAR f: File;  
                    VAR bufPtr: ARRAY OF BYTE;  
                    requestedBytes: LONGCARD;  
                    VAR read : LONGCARD);  
(* read an entire record from f *)  
  
PROCEDURE WriteRecord (VAR f: File;  
                    VAR bufPtr: ARRAY OF BYTE;  
                    requestedBytes: LONGINT;  
                    VAR written : LONGINT);  
(* write an entire record to f *)  
  
PROCEDURE Name (VAR f: File;  
              VAR filename: ARRAY OF CHAR);  
(* returns the name of the file currently open on f *)  
  
PROCEDURE FileNameLengthCheck (Filename : ARRAY OF CHAR;  
                              VAR resultname : ARRAY OF CHAR);  
(* Correction filenamelength *)  
  
PROCEDURE Parse (filename : ARRAY OF CHAR;  
                VAR resultname : ARRAY OF CHAR);  
  
PROCEDURE ShowStatus;  
(* should be called if not Done() to display  
  the corresponding error message on the terminal *)  
  
END FileSystem.
```

```
DEFINITION MODULE InOut;

  FROM FileSystem IMPORT File;

  CONST
    EOL = FileSystem.EOL;

  VAR
    Done : BOOLEAN;
    termCH : CHAR;
    in, out : File;

  PROCEDURE OpenInput(defnam: ARRAY OF CHAR);
  (* request a file name and open input file "in".
    Done := "file was succesfully opened".
    If open, subsequent input is read from this file.
    Name is parsed with defnam as default name *)

  PROCEDURE OpenOutput(defnam: ARRAY OF CHAR);
  (* request a file name and open output file "out".
    Done := "file was succesfully opened".
    If open, subsequent output is written on this file *)

  PROCEDURE CloseInput;
  (* closes input file; returns input to terminal *)

  PROCEDURE CloseOutput;
  (* closes output file; returns output to terminal *)

  PROCEDURE Read(VAR ch: CHAR); (* Done := NOT Eof(in) *)

  PROCEDURE ReadLn;
  (* skip to the beginning of the next input line *)

  PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
  (* read string, i.e. sequence of characters not containing
    blanks nor control characters; leading blanks are ignored.
    Input is terminated by any character <= " ";
    this character is assigned to termCH *)

  PROCEDURE ReadInt(VAR i: INTEGER);
  (* read string and convert to integer. Syntax:
    integer = ["+"|"-"]digit{digit}.
    Leading blanks are ignored.
    Done := "integer was read" *)

  PROCEDURE ReadCard(VAR c: CARDINAL);
  (* read string and convert to cardinal. Syntax:
    cardinal = digit{digit}.
```



```
    Leading blanks are ignored.
    Done := "cardinal was read" *)

PROCEDURE ReadOct(VAR c: CARDINAL);
(* read string and convert to cardinal. Syntax:
   cardinal = octdigit{octdigit}.
   Leading blanks are ignored.
   Done := "octal cardinal was read" *)

PROCEDURE ReadHex(VAR c: CARDINAL);
(* read string and convert to cardinal. Syntax:
   cardinal = hexdigit{hexdigit}.
   Leading blanks are ignored.
   Done := "hexa cardinal was read" *)

PROCEDURE ReadReal(VAR c: REAL);

PROCEDURE Write(ch: CHAR);

PROCEDURE WriteLn;

PROCEDURE WriteString(s: ARRAY OF CHAR);

PROCEDURE WriteInt(i: INTEGER;
                   n: CARDINAL);
(* write integer i with (at least) n characters on
   file "out". If n is greater than the number of
   digits needed, blanks are added preceding the number *)

PROCEDURE WriteCard(c: CARDINAL;
                    n: CARDINAL);

PROCEDURE WriteOct(c: CARDINAL;
                   n: CARDINAL);

PROCEDURE WriteHex(c: CARDINAL;
                   n: CARDINAL);

PROCEDURE WriteReal (r : REAL;
                    n : CARDINAL);

END InOut.
```

```
DEFINITION MODULE M2System; (* HS + WH 19.02.86 *)

  FROM SYSTEM IMPORT ADDRESS;

  (* 'System' FOR MC68000/MC68010 runtime support of Modula-2. *)

  VAR argc: LONGCARD;
      (* number of command-line arguments *)
      argv: ADDRESS;
      (* pointer to array of command-line arguments *)
      environ: ADDRESS;
      (* pointer to array of environment variables/values *)

  PROCEDURE HALTX;
  (* argument in register D0 ! *)

  PROCEDURE MULU32;
  (* arguments and quadword-result in regs. D0/D1 ! *)
  PROCEDURE DIVU32;
  (* arguments and quadword-result in regs. D0/D1 ! *)
  PROCEDURE MULS32;
  (* arguments and quadword-result in regs. D0/D1 ! *)
  PROCEDURE DIVS32;
  (* arguments and quadword-result in regs. D0/D1 ! *)

  PROCEDURE FADDs;
  (* (adder, addend : REAL) : REAL; *)
  PROCEDURE FSUBs;
  (* (minuend, subtrahend : REAL) : REAL; *)
  PROCEDURE FMULs;
  (* (multiplicand, multiplier : REAL) : REAL; *)
  PROCEDURE FDIVs;
  (* (dividend, divisor : REAL) : REAL; *)
  PROCEDURE FREMs;
  (* (dividend, divisor : REAL) : REAL; *)
  PROCEDURE FCMPs;
  (* (first, second : REAL); *) (* result in CCR *)
  PROCEDURE FNEGs;
  (* (toNeg : REAL) : REAL; *)
  PROCEDURE FABSS;
  (* (toAbs : REAL) : REAL; *)
  PROCEDURE FLOATs;
  (* (toFloat : LONGINT) : REAL; *)
  PROCEDURE TRUNCs;
  (* (toTrunc : REAL) : LONGINT; *)

  (* note that double precision arithmetic is not
     implemented yet *)
  PROCEDURE FADDd;
```

```
(* (adder, addend : LONGREAL)           : LONGREAL; *)
PROCEDURE FSubd;
(* (minuend, subtrahend : LONGREAL)     : LONGREAL; *)
PROCEDURE FMULd;
(* (multiplicand, multiplier : LONGREAL):LONGREAL;*)
PROCEDURE FDivd;
(* (dividend, divisor : LONGREAL)       : LONGREAL; *)
PROCEDURE FREMd;
(* (dividend, divisor : LONGREAL)       : LONGREAL; *)
PROCEDURE FCMPd;
(* (first, second : LONGREAL); *)(* result in CCR *)
PROCEDURE FNEGd;
(* (toNeg : LONGREAL)                   : LONGREAL; *)
PROCEDURE FABSD;
(* (toAbs : LONGREAL)                   : LONGREAL; *)
PROCEDURE FLOATd;
(* (toFloat : LONGINT)                   : LONGREAL; *)
PROCEDURE TRUNCd;
(* (toTrunc : LONGREAL)                  : LONGINT; *)

PROCEDURE FLONG;
(* (toConvert : REAL)                    : LONGREAL; *)
PROCEDURE FSHORT;
(* (toConvert : LONGREAL)                : REAL; *)

END (* OF DEFINITION MODULE *) M2System.
```

```
DEFINITION MODULE Procs;

  FROM SYSTEM IMPORT ADDRESS;

  TYPE
    PROCESS = ADDRESS;
    SIGNAL = POINTER TO ProcessDescriptor;
    (* the object coroutines synchronize with *)
    ProcessDescriptor =
      RECORD
        next: SIGNAL;
        (* linked list of active coroutines *)
        queue: SIGNAL;
        (* queue of coroutines waiting on
           specific signal *)
        cor: PROCESS;
        (* control block maintained by the system *)
        ready: BOOLEAN;
        (* flag for ready or blocked state *)
        paddr: ADDRESS;
        (*address of low-level process control block*)
        psize: CARDINAL;
        (* size allocated to process *)
        priority: INTEGER;
        (* user-assigned priority
           - main program always = -1 *)
        main: BOOLEAN;
        (* indicates whether is main coroutine
           or child *)
      END;

  PROCEDURE StartProcess (P: PROC;
                        worksizes: CARDINAL;
                        pri: INTEGER);

  PROCEDURE StopProcess;

  PROCEDURE SEND (VAR sendsig: SIGNAL);

  PROCEDURE WAIT (VAR waitsig: SIGNAL);

  PROCEDURE Awaited (signal: SIGNAL): BOOLEAN;

  PROCEDURE Init (VAR signal: SIGNAL);

END Procs.
```

```
DEFINITION MODULE Storage;

  FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE ALLOCATE(VAR p: ADDRESS;
                    size: CARDINAL);

  PROCEDURE DEALLOCATE(VAR p: ADDRESS);
  (* (size: CARDINAL); I took off this parameter *)

  PROCEDURE available(size : CARDINAL) : BOOLEAN;
  (* Return TRUE if space is available *)

  PROCEDURE REALLOCATE(VAR a : ADDRESS;
                      size : CARDINAL);
  (* Change the size of the previously allocated
   block pointed to by "a" to now contain "size"
   bytes. The address of the new block is returned
   in "a". *)

END Storage.

DEFINITION MODULE Strings;

  PROCEDURE StringLen(Str : ARRAY OF CHAR) : INTEGER;
  (* StrLen returns the length of the string Str *)

  PROCEDURE CopyString(Str1 : ARRAY OF CHAR;
                      VAR Str2 : ARRAY OF CHAR);

  PROCEDURE AppendString(Str1, Str2 : ARRAY OF CHAR;
                        VAR Str3 : ARRAY OF CHAR);

  PROCEDURE CmpString(Str1, Str2 : ARRAY OF CHAR)
                    : INTEGER;
  (* 0 when Str1 and Str2 are identical;
   -1 if Str1 lexicographically < Str2;
   1 if Str1 lexicographically > Str2; *)

END Strings.
```

```
DEFINITION MODULE Terminal;

PROCEDURE Read(VAR ch: CHAR);

PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
(* read a line of characters from the current *)
(* position until a line terminator is detected *)
(* or the string is complete filled *)

PROCEDURE ReadLn;
(* skips to the beginning of the next line *)

PROCEDURE ReadAgain;

PROCEDURE Write(ch: CHAR);

PROCEDURE WriteString(s: ARRAY OF CHAR);

PROCEDURE WriteLn;

END Terminal.
```

```
DEFINITION MODULE MathLib0;
(* This library module is modified from the C math
 * library. Due to the fact that the Modula II compiler
 * is limited to only 32-bit calculations, this math
 * library module will only return numbers with the
 * precision of 7 significant digits. *)

TYPE
  MathLibStatus = (Valid, InValid);

VAR
  MathStatus : MathLibStatus;

PROCEDURE sqrt(x : REAL) : REAL;
(* Return the square root of x, if x < 0.0 then
  return 0.0 *)

PROCEDURE exp(x : REAL) : REAL;
(* This exponential function returns e**x, if underflow
 * then return 0.0 else if overflow then return MaxFloat *)

PROCEDURE ln(x : REAL) : REAL;
(* Return natural log of x, if x <= 0.0 then return
 * LnMinReal *)

PROCEDURE arctan(x : REAL) : REAL;
(* x is in radian instead of degree *)
(* Return the arctangent of x. *)

PROCEDURE sin(x : REAL) : REAL;
(* x is in radian instead of degree *)
(* Return the sine of x. *)

PROCEDURE cos(x : REAL) : REAL;
(* x is in radian instead of degree *)
(* Return the cosine of x. *)

PROCEDURE pow(x, y : REAL) : REAL;
(* Return x to the power y, x**y, if underflow then
 * return 0.0 else if overflow then return MaxFloat *)

PROCEDURE entier(x : REAL) : INTEGER;
(* Return the nearest INTEGER to the REAL x. *)

PROCEDURE real(a : INTEGER) : REAL;
(* Return the REAL representation of the INTEGER x. *)

END MathLib0.
```

Implementing Run-Time Support for Modula-2

by

Wai-Sum Christopher Li

B.A. Ottawa University, 1986

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

Abstract

The goal of this project is to make the computer programming language, Modula-2, available on the UNIX PC for research and academic purposes. It involves writing the library modules and compiling them on the AT&T 3B15, then porting them to the UNIX PC for testing. The library modules are categorized into Run-Time library and Utility library. The former is written in C to support the execution of the Modula-2 compiler on the UNIX PC. The Utility library modules are written in Modula-2 except the coroutine primitives in Assembly of MC68010. A discussion on the standardization of Modula-2 libraries from the British Standard Institute is also included and compared with our implementation on the UNIX PC.