1984

Modula-2 concurrency

Philip E. Rosine
The University of Montana

Follow this and additional works at: https://scholarworks.umt.edu/etd
Let us know how access to this document benefits you.

Recommended Citation
https://scholarworks.umt.edu/etd/5540

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.
COPYRIGHT ACT OF 1976

THIS IS AN UNPUBLISHED MANUSCRIPT IN WHICH COPYRIGHT SUBSISTS. ANY FURTHER REPRINTING OF ITS CONTENTS MUST BE APPROVED BY THE AUTHOR.

Mansfield Library
University of Montana
Date: 1984
Modula-2 Concurrency

by

Philip E. Rosine
B.S., University of Idaho, 1969

Presented in partial fulfillment
of the requirements for the degree of
Master of Science
UNIVERSITY OF MONTANA
1984

Approved by

Chairman, Board of Examiners

Dean, Graduate School

Date

May 31, 1984
Modula-2 Concurrency

Advisor: Dr. John R. Barr

Modula-2, a language developed by Niklaus Wirth, is particularly well suited for teaching computer programming. Since it allows use of low level machine facilities it can be used for experimentation in many areas of computer science. One particular area of interest is that of concurrent programming, in which several parts of a program are executed simultaneously. Several methods and programming constructs for concurrent programming have been suggested and are of interest to the student. This paper deals with the development of a system in Modula-2 which is primarily designed for classroom use in teaching concurrent programming concepts and techniques.

Since the available computer is a PDP-11/23, a single processor system, a time-sharing concurrency scheduler was developed to simulate concurrent execution of processes. Semaphores were implemented as control structures for concurrent programs. The scheduler package includes a trace feature which allows a programmer to follow the execution of a concurrent program. A pre-processor to convert Modula-2 modules into monitors has been developed. This allows programming of concurrent programs using higher level constructs than are allowed by using the scheduler alone.
<table>
<thead>
<tr>
<th>Abstract</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Project Formulation</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Overview of the Problem</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Components of a Teaching Package</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Methodology</td>
<td>6</td>
</tr>
<tr>
<td>1.4.1 Analysis</td>
<td>7</td>
</tr>
<tr>
<td>1.4.2 Design</td>
<td>8</td>
</tr>
<tr>
<td>1.4.3 Implementation</td>
<td>8</td>
</tr>
<tr>
<td>1.4.4 Analysis</td>
<td>9</td>
</tr>
<tr>
<td>1.4.5 Documentation</td>
<td>9</td>
</tr>
<tr>
<td>1.4.6 Project Review</td>
<td>9</td>
</tr>
<tr>
<td>1.5 Proposed Work Plan</td>
<td>10</td>
</tr>
<tr>
<td>1.6 Preliminary Analysis</td>
<td>12</td>
</tr>
<tr>
<td>2 Concurrency — an Overview</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Problems Associated with Concurrency</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Proposed Solutions</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Co-routines</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Semaphores</td>
<td>20</td>
</tr>
<tr>
<td>2.5 Monitors</td>
<td>21</td>
</tr>
<tr>
<td>2.6 Message Passing</td>
<td>22</td>
</tr>
<tr>
<td>2.7 Rendezvous</td>
<td>22</td>
</tr>
<tr>
<td>2.8 Path Expressions</td>
<td>23</td>
</tr>
<tr>
<td>2.9 Communicating Sequential Processes</td>
<td>24</td>
</tr>
<tr>
<td>2.10 Eventcounts and Sequencers</td>
<td>24</td>
</tr>
<tr>
<td>3 Phase One — the basic modules</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Input/Output</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 Module InOut</td>
<td>28</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Monitor Algorithm Table ................................ 61
LIST OF ILLUSTRATIONS

Level 0 Data Flow Diagram ........................................... 12
Level-1 Data Flow Diagram ........................................... 52
Level-2 Data Flow Diagram ........................................... 57
Monitor Pre-Processor Design ........................................ 66
Monitor Pre-Processor Implementation .............................. 75
Chapter 1  
Project Formulation

1.1. **Background**

The primary rationale for this project is that multi-tasking and multi-processing are becoming increasingly important to applications programming on computers. In the past these topics and their associated problems have been primarily the concern of specialists dealing with operating systems and computer design, and for systems analysts on larger mainframe computer systems. Two of the current trends in computer science, both in terms of research and in terms of application, are distributed processing systems and multi-tasking programming languages. In order to better prepare students for these challenges, a programming language which allows
easy implementation of concurrent processes[1] and the associated control/data structures is needed. One such language is Modula-2, a computer programming language developed by Niklaus Wirth.

Modula-2 is descended from the languages Pascal [Jensen,1978] and Modula [Wirth,1977], which were also developed by Wirth. It is essentially an extension of Pascal to include the concepts of modules and multiprogramming. The module concept implies both the idea of separate compilation units for program parts, and the concept of information hiding. There are essentially three types of modules: program modules, definition modules, and implementation modules. A program module, as the name implies, is the main program, and only one such module will exist for any Modula-2 program. The definition and implementation modules come in pairs: the definition module states what data elements and procedure names the module can export to other parts of the program, and the implementation module contains the actual code which 'implements' these exportable pieces. This means that anything not defined as exportable is totally hidden from the user of the module (ie, another

[1] Definition: concurrent processes ==> more than one process executing in the same time frame, although perhaps not simultaneously. See Chapter 2 for a further discussion of concurrency.
module, or the programmer); information which is 'hidden' can be changed as desired (as long as the definition does not change) within the implementation module without affecting the user. It also means that a 'user' cannot accidently change the hidden portions of the implementation.

The concept of multiprogramming is also very important for teaching. Many languages currently being used do not directly allow the programmer to work with multiple programs as concurrent processes, although such languages are becoming more common. Co-routines, where control is passed back and forth between two or more routines, are the basis of Modula-2 multiprogramming. Co-routines, however, are not actually concurrent processes. The ability to use co-routines does allow the programmer to implement a scheduler process which will simulate concurrency (see Chapter 2). Modula-2 also contains the capability for the programmer to use low-level routines to interact with external interrupts and device controllers, which could be used to implement true concurrency. These features make Modula-2 an excellent learning tool.
1.2. Overview of the Problem

The Modula-2 version[2] available in the Computer Science Department contains some compiler bugs, and has several features which make experimentation with concurrency difficult. These problems include:

# Compiler errors which cause the compiler to abort without listing errors or to enter non-interruptable infinite loops.

# The system we have does not exactly match the system described by Wirth in his book [Wirth,1983].

# Difficulty in starting concurrent processes, and in getting output from them.

# Lack of a 'trace' feature which would allow the novice to easily experiment with concurrency.

Eliminating these problems would greatly enhance the education facilities available for students who wish to study the problems of concurrency. The overall thrust of my proposed graduate project will be to design and implement a package of modules which can be used to

allow students to experiment with concurrent processes easily and (more or less) efficiently. This will require first correcting the deficiencies of the current Modula-2 implementation.

1.3. Components of a Teaching Package

This project will include the design and implementation of a package of modules for use in teaching the theory and application of concurrent processes and concurrent programming. Modules need to be written to implement message-passing, communicating sequential processes, and the rendezvous (see Chapter 2 for a detailed explanation). These concepts are very important for training in the use of concurrent processes. It will not, in general, be feasible to implement these as they were originally specified, since they really require programming languages (compiler, etc.) of their own. It will, however, be possible to write a preprocessor which will translate non-Modula-2 statements, such as Hoare's [1978] Communicating Sequential Processes syntax, into Modula-2 calls which can then be compiled using the standard Modula-2 compiler.

A trace option which would allow a programmer to follow the actual execution of his program is also a desirable
feature. Concurrency implies some degree of indeterminacy for the processes involved and their interactions. This indeterminacy (i.e., the exact relationship of command execution between the processes is unknown and unknowable until the program is executed) is the underlying problem in all multiprogramming. Technical methods have been proposed for formally proving concurrent programs [Taylor, 1983], but such methods are difficult to use and limited in scope. This means that rigorous testing may be the only means of determining program correctness in some cases. Being able to trace the program execution without extra coding would be a valuable tool for the programmer, and particularly for student programmers. It would both allow better testing of programs and give the student better insight into the concept of concurrent processes.

1.4. Methodology

The several problems noted above appear to be more or less independent, although there may be some overlap which will not be apparent until I begin to seriously analyze the current Modula-2 implementation. Because of this independence, I propose to use a modified form of incremental development for this project, concentrating
on each of the indicated problem areas in turn. Modula-2 lends itself well to this type of plan since its structure allows modules to be replaced or added without affecting the rest of the system. Incremental development means that each phase of the project includes the software engineering processes of analysis, design, implementation, and testing. Several techniques and tools will be used to accomplish this development.

1.4.1. Analysis During the analysis phase, DeMarco's structured analysis methodology [DeMarco, 1978] will be primarily used. It will be supplemented by use of PSL/PSA (Problem Statement Language / Problem Statement Analyzer -- a computerized analysis tool developed by ISDOS at the University of Michigan). Structured analysis allows accurate and relatively easy specification of the exact problem to be solved, including the processes and data needed for solution of the problem. It does not actually design a solution to the problem, but does make design much easier. The use of PSL/PSA as a part of structured analysis allows much of the documentation to be done on the computer using a specialized data-base. This further allows use of the computer for analysis of the problem for completeness and accuracy using special programs supplied by PSA. An additional advantage is that this method and the
supporting tool improves the overall documentation of the system. Whenever possible, the writing of user manuals will be done as a part of the analysis phase (or during the initial design phase). Doing the manual writing early in the development sequence has several advantages: it allows checking of the proposed problem statement and solutions against the actual problem (by the end user), it gives the design something concrete to be measured against, and it solves some of the historical (traditional) problems of incomplete and inaccurate documentation.

1.4.2. Design The design phase will follow (generally) the method of Yourdan and Constantine [Yourdan, 1979]. The System Encyclopedia Manager (SEM, from ISDOS) will be used to document the design. Use of this methodology and the Program Design Specification Language (PDSL) (which works with SEM) have similar advantages to those described in the previous section.

1.4.3. Implementation Implementation will obviously be done using Modula-2. Some of the advantages of using Modula-2 were described in a previous section. Additionally, all code will be documented (possibly by reference to the documentation provided by the design phase). If required, some portions of the code may
have to be implemented in Macro-11 in order to interface with the hardware.

1.4.4. Testing  Testing will be done using test cases from Wirth [1983], or with program solutions provided by the authors of the various papers used as primary sources. Where required, additional test cases will be developed (in general, testing for this project means writing and executing Modula-2 programs).

1.4.5. Documentation  In all phases documentation will be maintained using the University of Montana DECsystem-20 and the Computer Science Department UNIX system. As indicated previously, documentation must be an ongoing process, and should be up to date at the end of each phase of development. When the project is finally tested and accepted, it should be complete.

1.4.6. Project Review  It is desired that progress be regularly reviewed and critiqued (using a modified form of the structured walk-through method [Basili,1981]) by the whole committee. Ideally, a walk-through involves peer presentation and evaluation, but the technique can be used for the current situation. In most cases, a preliminary walk-through will be held by the Graduate Advisor. Prior to the committee meeting, the material to be reviewed will be distributed so that members may
familiarize themselves with it. The walk-through it itself involves a brief presentation of the material and critique by the committee. It is primarily an error detection session, although suggestions for solution, correction, and improvement are appropriate for this type of modified walk-through.

1.5. Proposed Work Plan

The following phases and schedule are proposed:

Phase One -
Develop test cases to demonstrate the known and suspected compiler bugs using the current implementation. The test cases developed using the current compiler will be used to verify the new version. The compiler should be capable of running all of the example programs given in Wirth's book. Phase One should be completed by the end of Fall Quarter 1983.

Phase Two -
Develop an implementation package for concurrent processes and I/O. This package will require further specification before beginning the design and implementation phases. The concurrency package will consist of modules which can be included by
the programmer (ie, they will not be integral to the Modula-2 compiler or the run-time system). The package will include preprocessors to implement the syntax for some of the concurrent programming concepts such as the rendezvous. Phase Two should be completed by about the middle of Winter Quarter 1984.

Phase Three -
Implement a trace feature for Modula-2 programs. This could take one of several forms: it could be automatically generated code handled by a compiler switch; it could be code added by a preprocessor; it could be handled by the Modula-2 run-time system; or it could be a set of modules which a programmer could use (via IMPORT) as desired. Further analysis and specification will be necessary for this, and I feel that a decision about this should not be made until the compiler has been examined and fixed. This phase should be complete by the end of Winter quarter 1984 or early in Spring quarter.

This schedule will allow this package to be used during Spring quarter 1984 if a class in concurrent programming is taught again. This would act as a rigorous test of
my work, as well as benefiting the class directly.
Also, this schedule allows time for me to carry out some
additional work in developing a major example program
using Modula-2. Such a program, essentially a 'coda' to
the project, would probably take the form of a model
multi-tasking operating system (possibly a multi-
processor system).

1.6. Preliminary Analysis

Following this proposal is a top-level (Level-0) data-
flow diagram of the overall project environment. A data
dictionary in the form of a PSL/PSA formatted problem
statement is included in Appendix D. The actual project
'space' only includes the Compiler/Linker, the Run-
Time-System, the SubRoutine-Library, and the Compiler-
Library. This preliminary analysis therefore shows more
detail than is theoretically needed, but due to the
interaction of the 'pieces' of the project with the
'real world' I feel that a broader scope must be
detailed during the initial analysis phase.
Modula-2 System Overview
Chapter 2
Concurrency -- an Overview

The term 'concurrent processes' was earlier defined as "more than one process executing in the same time frame, although perhaps not simultaneously". This definition is obviously incomplete, for it leaves undefined several other critical terms. This section will present a glossary of some of the terms used in discussing concurrency, and will then briefly present a discussion of the problems presented by concurrent processes and their solutions.

concurrent

Happening at essentially the same time. Concurrent as applied to computer processes does not imply that things happen at exactly the same time. Two modifiers are used with 'concurrency' to better
indicate the relationship.

true (or real) concurrency
Events are simultaneous (ie, they happen at exactly the same instant of time). For computers this requires multiple CPUs (central processing units) (ie, a multi-processor system).

apparent concurrency
Actions appear to be concurrent over some short span of time; for computers this implies time-sharing on a single CPU. For practical purposes it does not matter whether concurrency is real or apparent: the uses and problems associated are the same.

interleaving
This refers to the way the scheduler starts, stops, and re-starts processes on a time-sharing system. The execution of the processes is said to be interleaved since P1 is followed by P2 which is followed by P1, etc.

multi-processing
Allowing execution of several processes at once; either real or apparent concurrency.
multi-processor

Having more than one CPU for a single operating system. The memory used by such a system may be either shared or separate (ie, different memory for each CPU), or some combination.

multi-programming

This term is somewhat confusing. Essentially it means that more than one program at a time can be in a state of execution. Wirth [1983] uses multi-programming to indicate any form of concurrency, including the case where a single program is being executed, but several parts of the program are executing concurrently. See multi-tasking.

multi-tasking

Multi-tasking is often used more or less synonymously with multi-programming. A better use would be to consider a task as a subpart of a program, and use multi-tasking for the situation where a program executes several of its parts concurrently. This implies that 'program' should be redefined as a technical term to indicate a set of instructions with a particular form which allows completely independent execution. A task does not have this character of independent execution:
instead a parent-child relationship exists between a program and its tasks such that completion of the program requires completion (perhaps premature) of all tasks, but not the reverse.

**process**

A program in a state of execution.

**program**

A set of executable instructions for the CPU. A program may be executed as a single process, or subsets of the 'program' may be executed as several processes. For most purposes, the term program will refer either to source code instructions (which may not be executable without further processing) or machine executable instructions. Essentially, a program (or the process resulting from its execution), causes the computer to accept some input data and transform that input into some output data. Note that a program and a process are related but very different things: a program (in the computer science sense) is a collection of data items which make up instructions to a machine (computer) telling it how to accomplish some function; a process is a program in a state of execution. This means that a process is actively doing some-
thing, while a program is a static set of data. See multi-tasking.

time-sharing
This implies that several processes execute partially during some given period of time (usually some small fraction of a second). At any given instant, however, only one process is executing. Some sort of supervisor or scheduler is required to switch processes in and out of a state of execution.

2.1. Problems Associated with Concurrency

The two principle problems which must be addressed when working with concurrent processes are mutual exclusion and synchronization. Mutual exclusion involves the difficulties which surround the use of some critical resource (say a lineprinter) by two (or more) processes which are being executed concurrently. Synchronization becomes a concern when two (or more) concurrent processes must carry out some actions in a particular sequence (e.g., a character must be added to a buffer before it can be processed). A number of 'classic' problems have been presented in the literature which illustrate these situations [Ben-Ari,1982]. Likewise, a
number of solutions to these problems have been sug-
gested.

2.2. Proposed Solutions

There are basically two ways of handling the problems of
concurrency: use some form of shared data which all
concurrent processes can access without interfering with
each other, or pass data between processes without the
data being 'shared' in the sense that a single memory
location is shared. The idea of using shared data items
has led to the semaphore and monitor concepts for con-
trolling concurrency. The use of passed data has led to
the several forms of message passing (as in the Ada ren-
dezvous and the communicating sequential processes syn-
tax). Both of these schemes have advantages and disad-
vantages which have been discussed in various articles.
[Stankovic,1982], [Eventoff,1980]. A brief discussion
of some of the solutions and the feasibility of includ-
ing them in a Modula-2 package follows.

2.3. Co-routines [Wirth,1983][3]

[3] References given in the section headings are either
to the original source document for the concept or to a
major reference for the general subject. In most cases
[Ben-Ari,1982] is also a useful reference.
The primary mechanism for multi-process interaction in Modula-2 is the co-routine. This essentially allows the programmer to deliberately switch between two (or more) routines, but does not really involve concurrency. To obtain concurrency, some mechanism is required which will have the CPU execute multiple processes in some time-slicing fashion or will start separate processes on separate CPUs. A supervisor which starts multiple processes in apparent concurrency is available on our system, but it is somewhat difficult to work with. This module may require some re-writing and the user documentation needs to be improved.

2.4. Semaphores

One of the first widely used methods for controlling concurrency was semaphores, which were first proposed by Dijkstra in 1968. A semaphore is really a data structure which may be used in a restricted manner (using functions P(s) and V(s) on semaphore s) to control access to critical resources or for synchronization. The semaphore itself is a shared data item, but access to it is restricted to the P and V operations, and these operations are constructed so that only one process at a time can use them, and this provides mutual exclusion for the semaphore. The semaphore can then be used to
provide mutual exclusion and synchronization for other data and processes. The P and V operations are often called WAIT and SIGNAL, terms which perhaps better characterize their function in English. Semaphores and semaphore operations are very low level compared to some of the other constructs discussed below, and most other concurrent programming structures can be constructed by use of semaphores. Conversely, semaphores can be constructed using higher level concurrency constructs, like monitors. Semaphores are not a part of Modula-2, but can be constructed using monitors (ie, modules with priority -- see the next section). (See [Ben-Ari,1982], chapter 4 for a more detailed discussion of semaphores.)

2.5. Monitors [Hoare,1974]

A monitor is a program structure which controls access to its procedures. Use of a monitor directly implements mutual exclusion, since only one call to a monitor is allowed to be active at one time. The monitor also allows synchronization by use of SIGNAL and WAIT operations (similar to, but not exactly like the P and V operations on semaphores). A monitor may contain data structures and procedures. In Modula-2, an implementation module may be assigned a priority, which allows it to be used very much like a monitor, since all calls to
the module's procedures will have the same priority, they may not interrupt each other.

2.6. Message Passing [Stankovic, 1982]

Message passing is somewhat different from the use of semaphores or monitors in that the processes make contact via some external mechanism. No synchronization of the processes is required. In many message passing schemes there is no guarantee that a receiving process ever gets the message, although some implementations may allow the programmer to require a sending process to wait until a message is received. The receiver is usually required to wait for a message, which does allow some synchronization to be programmed, but it is sometimes an awkward method. In a sense, the SIGNAL and WAIT operations of our Modula-2 implementation are a type of message passing. Message passing should be fairly easy to implement using Modula-2, and will probably be used to clean up the I/O problems associated with concurrent processes.

2.7. Rendezvous [Eventoff, 1980]

The rendezvous is a somewhat more high-level concept than the monitor, but has not been used much in the past. The programming language Ada is the first to make
extensive use of the rendezvous. The rendezvous is really a special kind of message passing, although it has some of the characteristics of procedure calls. Synchronization is a major factor in the rendezvous, making it different from basic message passing. The rendezvous also differs from the monitor and co-routine concepts in that both the calling and called processes in the rendezvous are active when the call takes place. For the monitor (co-routine), only the calling process is active prior to the call, and only the called process is active after the call. The rendezvous is not directly implementable in Modula-2, but by defining a set of modules and procedures, it should be possible to simulate the action of the rendezvous. A pre-processor can then be written to convert the Ada syntax into Modula-2 calls.

2.8. Path Expressions [Campbell] [Kolstad]

Path expressions are a syntactic construct which allows easy and direct specification of concurrent process interaction. They are primarily for controlling synchronization. Path Pascal has been developed as a language for using path expressions, and there is a pre-processing program available at the University of Montana for converting path syntax into semaphore syn-
2.9. Communicating Sequential Processes [Hoare, 1978]

Based upon the concept of guarded commands [Dijkstra, 1975], Hoare, using the idea that input and output should be primitive operations in a computer language, specified a syntax for communicating sequential processes. This syntax does not completely define a new computer language, but could be used as the basis of one. The overall concept is directly related to message passing with required synchronization. The syntax is very concise and is a good notation for discussing concurrency. Because the syntax is very different from that of Modula-2, it will not be directly implementable. The proper modules, data structures, and procedures will have to be defined in Modula-2, and then a pre-processor written to translate the guarded command syntax into Modula-2 declarations and calls.

2.10. Eventcounts and Sequencers [Dobrowolski]

Eventcounts and sequencers are concepts which have been proposed for controlling concurrency on multi-processor machines. For best implementation, they require special hardware consideration during the design of the computer and special machine instructions. They are not really
applicable to most standard computing environments today, but it may be interesting to attempt a simulation of them in Modula-2.
Phase One of the Modula-2 project was intended to find and correct any compiler bugs and to implement the basic peripheral modules specified by Wirth [1983] plus a time-sharing process scheduler. Only part of these objectives have been completed. The modules InOut and RealInOut have been implemented (very nearly as specified by Wirth), and a time-sharing concurrency scheduler has been designed and implemented (copies of the DEFINITION and IMPLEMENTATION modules are provided in Appendix A). User manuals for these features have been provided (see Appendix B). No work has been done on the compiler or run-time systems, and the module MathLibO (which provides REAL arithmetic functions) has not been implemented. A new version of Modula-2 was installed last fall, and testing shows that most of the bugs previously
detected (in the old version) have been corrected. There appear to be some serious problems with the way type REAL is handled by Modula-2 but since the project is running somewhat behind schedule, it has been decided not to try to fix these at this time. If time remains after the important parts of Phases II and III are finished, work will continue in this area.

3.1. Input/Output

Wirth does not include any input or output facilities in his specification of Modula-2, but he does include the primitives allowing access to external devices, and the supplied compiler and run-time system includes a set of low-level modules for I/O. Wirth [1983] specifies two modules, InOut and RealInOut, to supply the needed programmer interface to user terminals and files. These modules were not supplied with the Modula-2 implementation. A somewhat different version of InOut was supplied (and used by the compiler, etc.), but it does not meet the specification given by Wirth in his text. It is apparently an earlier version of the I/O procedures.

Module InOut, as specified by Wirth, gives the programmer the ability to open and close files, and read and write characters, strings, and numbers of type INTEGER
and CARDINAL. Module RealInOut is specified to supply facilities for reading and writing numbers of type REAL. After these were implemented, and as work progressed on the process scheduler, a need became apparent for some means of controlling process access to the I/O functions in order to prevent interference between processes during input and output. This led to the development of the module IOControl, which provides an easy way of controlling such interaction.

3.1.1. Module InOut

No real difficulties were encountered in changing the supplied version of InOut so that it met the specification. Some slight changes and additions were made to the module during the course of making the conversion where it was apparent that they would be generally beneficial. The biggest of these changes was to allow the procedure ReadString to actually read a string; Wirth's version and his specification only allowed reading of a single word (reading was terminated by any 'white' space). This change made the function of the procedure consistent with its name and with the generally accepted definition of a character string.

File handling was found to contain one minor error in Wirth's version: only one OpenInput and one OpenOutput
were allowed in a program because the stream was not released by the Close procedure. This error was corrected so that multiple Opens (followed by corresponding Closes) will work correctly. An added feature causes the OpenOutput procedure to rename an existing file with a 'Q*' extension (similar to the DECsystem-20) so that a back-up is formed. This helps prevent accidental loss of data files.

The numerical output procedures WriteInt, WriteCard, WriteOct, and WriteHex were changed so that asterisks (*) are written into the output field if the field is too small for the data. This is not called for by the specification, but helps prevent output errors and is similar to features in other languages.

3.1.2. Module RealInOut

No version of RealInOut was supplied by the system, so this module had to be created from scratch. The algorithms used are relatively simple, being standard division algorithms for obtaining the desired character string for output (and vice versa for input). Although this is not particularly efficient, it seems to be the best method unless they are coded in assembly language. The type REAL itself presented some problems during development of this module. These problems are
discussed in the section on Compiler Bugs.

For programming convenience, the ReadReal function reads its input using procedure ReadString from module InOut. This causes the reading of REALs to work somewhat differently than the reading of INTEGERS or CARDINALs. This does not, however, violate the specification. The differences are explained in the I/O User Guide.

3.1.3. Module IOControl

This module was added in order to simplify the programmer's job. It hides the semaphore and semaphore operations needed to allow concurrent processes to do I/O without interfering with each other. The DEFINITION module exports two functions, AcquireIO and ReleaseIO, and the IMPLEMENTATION module imports the type SEMAPHORE and its operations from module PreemptiveScheduler. The implementation is very straightforward and did not really involve any design other than that needed for any concurrent program. The use of this module is discussed in the I/O User Guide.
3.2. Concurrent Processes -- Module PreemptiveScheduler

Modula-2 does not provide the programmer with the direct ability to use concurrent processes. It does provide for the use of co-routines and access to hardware interrupts. Using a combination of these, a time-sharing process scheduler has been developed. It is loosely based upon the module Processes specified by Wirth [1983], and more directly upon an adaptation of that module done by Dr. Barr [module Scheduler, University of Montana, 1982]. Wirth's module Processes did not provide concurrency, it simply gave the user a higher level version of co-routines, and means to control interaction between co-routines. Dr. Barr's Scheduler module provided for true process scheduling, but not for real time-sharing. It did provide for transfer of control between processes based partially upon clock interrupts, but was much more rigid and determinate than a time-sharing scheduler.

A time-sharing system requires a preemptive scheduler, in which the current process will be interrupted by the system clock, and control transferred to another process (if available). This interrupt must be transparent to the ready processes; ie, they must neither know nor
care that it happens. The object is to simulate several sequential processes each running on a separate CPU in its own memory space. Interaction between the processes is (usually) necessary in order for meaningful work to be done, and the means of controlling this interaction have to be provided. The module PreemptiveScheduler provides this type of process scheduler.

3.2.1. Requirements & Specifications

Contrary to good practice, the requirements and specifications for this module were not formally set down prior to starting work on the implementation. The primary reason for this was that the original intent was simply to make minor modifications to the existing scheduler as supplied by Dr. Barr. As work progressed on this modification, it became apparent that major modification was taking place based upon an intuitive understanding of what the requirements for a scheduler should be. Much of the design and implementation phases thus converged, with the final module being the result of a rather hueristic (as compared to a strictly formal) design process. Formalization of the specification did not take place until the module was complete and tested.
The intuitive requirement for a concurrent process scheduler is that it be able to start and stop processes and provide for interaction between those processes. If it is to be implemented as a time-sharing system on a single CPU (as this one is), an additional requirement is for fairness of process interleaving. Fairness implies that all processes which are ready to run will have equal access to the CPU on a time-shared basis.

The requirement for starting processes is fairly obvious. The halting requirement is not quite so obvious: it seems intuitive that a started process must either run infinitely or it must stop at some point, but Modula-2 does not necessarily support this 'stopping' of processes directly. This requirement was added explicitly when testing showed that when a process is allowed to end by reaching a normal procedure end, the system will crash and require rebooting. Additionally, our implementation of Modula-2 (on the PDP-11) requires that when a program ends, the scheduling process must be explicitly halted. The requirements for process interaction control must be met by two sets of data types/procedures. One of these is like the SIGNAL type shown in Wirth's implementation; the other is a SEMAPHORE type. These must be included directly in the
scheduler module since they require access to data and procedures which logically and functionally must be hidden by the scheduler. The specification, then, is as follows:

S1.) Execution of concurrent processes:

S1.1.)
Must be able to start concurrent processes.

S1.2.)
Must be able to halt a process once started. When the program ends, the time-sharing scheduler must also be halted (either explicitly or implicitly).

S1.3.)
Time-sharing by ready processes must be provably fair.

S2.) Process interaction controls:

S2.1.)
Must provide an exported type SEMAPHORE with P and V operations on that type. Type SEMAPHORE should be a full-range counting semaphore as described by Ben-Ari [1982], and an operation on the type allowing initialization to any CARDINAL value must be supplied.
S2.2.)

Must provide an exported type SIGNAL as defined by Wirth [1983] in his Processes module. Operations similar to those shown for this type must be provided.

3.2.2. Design & Implementation

As stated previously, the design of this module is largely heuristic. It was developed over a period of about a month by means of a series of tests and trials. Consideration was always given to the design concepts of cohesion and connectedness; ie, at all stages attempts were made to keep the module as functionally consistent as possible (both as a whole and within its component parts) and to minimize the interface to other parts of the system. The design progressed by testing various procedures and ideas separately to find out how they worked, and then deciding how to combine them to meet the specification. Much too frequently, certain things (particularly the system calls TRANSFER and IOTRANSFER) did not work quite the way they were expected to work. This meant that the design had to be re-thought and the implementation reworked and re-tested.
The PreemptiveScheduler module is relatively straightforward once it is understood, but the implementation may be somewhat confusing at first sight. Before attempting to follow the implementation, the reader should study the User's Guide, and if possible write some concurrent programs using the scheduler. The following discussion will attempt to follow the workings of the scheduler in a logical fashion from the point of view of an actual program sequence. The listing of the IMPLEMENTATION module in Appendix A should be used to follow the discussion. The constants, variables, and data types will be discussed first. Then each procedure will be discussed as it appears in the progression through the module's activity.

An important point to be noted in discussing the scheduler's actions is its priority level. The module runs at priority 6, which is the same priority as the hardware clock interrupt. This means that procedures within the module are similar to those of a monitor, and cannot be interrupted by the clock interrupt. Thus time-sharing will not cause adverse effects upon the action of the scheduler itself. This does have some important side effects, though, particularly when process tracing is considered. At one point an attempt was made to integrate a trace feature into the
scheduler. This is not feasible since I/O operates at a lower priority than the clock, and calls to I/O routines are thus interruptable by the clock. The result is a loss of accuracy in the trace function, and confusion of the output.

3.2.2.1. Module Data Structures

The data structures used by the module are relatively simple. They are concerned primarily with setting up and controlling the action of the clock interrupt process.

The constants are for two purposes: to give names to the clock interrupt vector, and to give names to the return codes for the scheduler. Their action and use are self-explanatory.

Four data types are declared. Two of these are exported, and their function is to allow easy access to the other two types. Type SIGNAL is a pointer to a ProcessDescriptor, and is an important type in its own right. Exportation of the SIGNAL is done only to allow it to be used as a process synchronization tool (see procedures signal and wait). This type was the only process control element used by Wirth's Processes module. Type ProcessDescriptor is a record type; it
allows the scheduler to keep track of the processes being run. Each process has a ProcessDescriptor. The first item in the ProcessDescriptor, pp, is the PROCESS used by the system calls NEWPROCESS, TRANSFER, and IOTRANSFER. The second item, proc, is where the PROC (parameterless procedure) which is to be executed is stored. This is used by procedure job to run the desired procedure. ID is a variable added to the ProcessDescriptor for the purpose of allowing easy tracking of the process interaction by a user program.

ready is used to keep track of the state of the process. A process can be in one of three states: running, ready to run, or waiting (for something). Only two of these are reflected by ready: if ready is TRUE, then it can be run when its turn comes up in the scheduling queue; if ready is FALSE, then the process is in a wait state and will be skipped when it comes up in the queue. next defines the ready queue for processes. All processes which have been started and not halted are tied together in a loop (circular queue) by means of the pointer next in each ProcessDescriptor. Use of a loop satisfies the fairness requirement (SI.3.). The final item, queue, is used to link the processes waiting for the same SIGNAL or SEMAPHORE. This forms a FIFO queue, which satisfies
the requirement for fairness of a SEMAPHORE (implicit in the definition of a semaphore). When the exported SIGNAL is used for process synchronization, the variable declared in the user program is used as a pointer to the process which is waiting on the SIGNAL.

The other exported type is SEMAPHORE. This type exists only for the purpose of exportation; Modula-2 does not allow exportation of a hidden complex type. Type SemaphoreDescriptor is really the working portion of the semaphore implemented by this module. In order to hide the implementation as much as possible, it is not desirable to export this type in a direct fashion (by declaring it in the DEFINITION module). Therefore the exported SEMAPHORE is a pointer to the descriptor. The SemaphoreDescriptor consists of a counter used to keep the value of the semaphore, and a SIGNAL semqueue which is used to point to the first process in a FIFO queue of processes waiting for the semaphore.

The variable cp is used to point to the ProcessDescriptor of the process which is currently running. Variable main is used to keep track of the process for the program's main code. The two variables of type PROCESS are used by TRANSFER and IOTRANSFER in the operation of the scheduler. StorageForMain pro-
vides space for a ProcessDescriptor for main. WSP
provides workspace for a NEWPROCESS for the clock
interrupt PROCESS. Variable PCount keeps track of how
many processes have been started and not halted. Pnum
counts all process which have been started; the ID in
the ProcessDescriptor is assigned the current Pnum
when a process is started. Variables clktrap,
clkvect, savtrap, and savvect are used for saving and
restoring the system value of the clock interrupt.
running is used to indicate whether the scheduler
interrupt is operational or not; schedStatus indi­
cates what the current scheduler status is, and why
(if) it was halted. schedStatus and running are in
some sense redundant, but it seems cleaner to use both
of them.

3.2.2.2. Procedure start

Procedure start is the beginning point for the whole
scheduler process, as well as the beginning for each
individual process. When a program importing Preemp­
tiveScheduler begins, the main code for the module is
executed. The current (ie, the system) clock inter­rupt trap and vector are saved (on the PDP-11 these
are at addresses 100B and 102B). running is set to
FALSE, since the time-sharing clock is not started
until it is needed. schedStatus is set to InitState, and PCount and Pnum are both initialized to zero. If (and only if) running is FALSE when start is called (indicating that the clock interrupt is not active), the procedure InitScheduler is called.

InitScheduler sets up a ProcessDescriptor for the program's main line of code (it may actually be in a procedure at this point, but there are no other concurrent processes yet running). InitScheduler also gets a PROCESS for the procedure Clock. start then sets up the ProcessDescriptor for the procedure which is to become a process, and calls NEWPROCESS to get a PROCESS for it. The actual process is the procedure job (local to PreemptiveScheduler), which will call the desired procedure. The space for the ProcessDescriptor is reserved within the workspace before the call to NEWPROCESS. Failure to reserve this space and send the correct address and size to NEWPROCESS can cause unpredictable program execution if the system uses the ProcessDescriptor space as part of the stack since important data can thereby be lost. Setting up a ProcessDescriptor is relatively simple, involving putting the ProcessDescriptor address in the next loop, setting ready to TRUE, assigning an ID, and making queue NIL. At the end of start, if the running
is FALSE, a TRANSFER to the clock process (clk) is done, otherwise a normal return to the calling process occurs. Procedure start satisfies specification clause Sl.1.

2.2.2.1. Procedure job

This procedure is the result of a suggestion by Dr. Barr. Originally the scheduler exported procedure HaltProcess, and the programmer was required to explicitly code a HaltProcess at the end of any procedure (unless the procedure contained an infinite loop). After some experimentation, Dr. Barr recommended that the procedure (type PROC) which was to be executed be stored in the SIGNAL data structure, and a local procedure be used to call it. The local procedure can then call HaltProcess when (if) the called procedure ends. Procedure job does this. It is run as a process by the scheduler, and serves the purpose of hiding the HaltProcess call.

2.2.2.4. Procedure Clock

The procedure Clock is the clock interrupt process. This procedure is never called as a procedure, instead it begins through the TRANSFER from start (only when running is FALSE). When this TRANSFER occurs, running
is set to TRUE and schedStatus is set to RunState.
The process then enters an infinite loop, which immediately does an IOTRANSFER. The IOTRANSFER has the effect of setting the clock interrupt vector so that when the system (hardware) clock next 'ticks' (every 1/60th of a second) control will be transferred to the next statement in Clock (following the IOTRANSFER). Actual execution returns to whatever process in currently pointed to by PROCESS intbyclock instead of continuing procedure Clock. When the initial call to start is made, this process will be the main program (this was determined by the TRANSFER call in start), which is currently in procedure start (and a normal return from start occurs).

From this time on, until a call to HaltScheduler is made, every time the clock 'ticks', an interrupt will occur, transferring execution to the procedure Clock. Within Clock, the circular queue (item next in the ProcessDescriptor) is checked to find the next process which is ready to run. If no such process is found, then HaltScheduler is called (to prevent an infinite wait state error — see procedures wait and V). When a ready process is found, the PROCESS values for intbyclock are swapped, first to make the current process (cp^.pp) have the correct pointer for its next
execution cycle, and then to put the pointer for the next ready process in cp. The new cp^pp is moved to intbyclock, and the loop goes back to the IOTRANSFER, where control is transferred to the new process.

3.2.2.5. Procedure HaltProcess

HaltProcess is used to stop a process which has been started. It makes ready FALSE for the current process (cp) and decrements the count of runnable processes. If the halted process is the only running process other than the main program, schedStatus is set to indicate this occurrence, and HaltScheduler is called to stop the clock interrupt. If the scheduler is running, a TRANSFER to procedure Clock is made. This TRANSFER acts like a clock interrupt (described above), passing execution control to the statement following the IOTRANSFER call in Clock. This causes the scheduler to switch to the next ready process. If by some strange error the scheduler is not running, nothing is done. Together with procedure HaltScheduler, HaltProcess satisfies specification clause Sl.2.
3.2.2.6. **Procedure HaltScheduler**

The essential action of HaltScheduler is to reset the clock interrupt trap and vector to their original values. It also sets running to FALSE and sets the schedStatus value to reflect its action. If the current process is not the main program, it then TRANSFERS control to the main program. If the current process is the main, then a normal return is allowed.

HaltScheduler has no effect if the scheduler is not running (running equals FALSE); this can occur because the scheduler can be halted by call from HaltProcess or by Clock, and in some cases, the indeterminacy of the concurrent process interaction may cause a programmer to be unsure of when (and where) the halt will occur. Since multiple calls of HaltScheduler will have no adverse effects (but lack of such a call will have adverse impact), the programmer can thus safely ensure that the scheduler is halted when absolutely required. The reason for not executing HaltScheduler more than once is because testing indicated that resetting the clock interrupt vector too many times had adverse effects; the real cause of this is unknown.
3.2.2.7. Function StatusOfScheduler

Since the scheduler cannot use I/O operations directly, it must export its status and allow the user program to handle error output. This may be important if an application may result in all processes ending in a wait state. When this occurs, HaltScheduler will be called and control returned to the main program; the programmer needs to check schedStatus by calling StatusOfScheduler and outputting an error message if the return code is not as expected. This may also be useful for tracing program execution.

3.2.2.8. Function ProcessID

This function returns the value of ID for the current process, and allows the user program to trace its execution.

3.2.2.9. Operations on type SIGNAL

Type SIGNAL is used primarily for process synchronization. See the User's Guide for further discussion of its use. The SIGNAL, as discussed previously, is pointer to a ProcessDescriptor. As a user variable, the pointer is used to point to the first process in a FIFO queue of processes waiting on the particular SIGNAL. The queue is linked through the queue item in
the ProcessDescriptor. Procedure initsignal simply sets the pointer to NIL, forming an empty queue. Procedure wait puts a process in the queue, and sets its ready state to FALSE; if the scheduler is running, a TRANSFER to Clock causes the next ready process to start running. If all processes are in a wait state, Clock will call HaltScheduler to prevent a deadlock condition. Procedure signal changes the ready state of the next process in the queue (if any) to TRUE and moves the following process in the queue forward. This use of a FIFO queue insures fairness for processes using SIGNALS: a process waiting on a SIGNAL is guaranteed eventual return to the run state provided there are enough signals prior to program end. Function waited returns a BOOLEAN indicating whether or not any process is waiting on a particular SIGNAL. All of implementations are quite simple. This satisfies specification clause S2.2.

3.2.2.10. Operations on type SEMAPHORE

A SEMAPHORE is somewhat more complicated than a SIGNAL, since the actions defined for semaphores (generally) depend upon the current value of the semaphore. A SEMAPHORE is a complex type, and thus is not exported directly. initsemaphore therefore has to
allocate space for the SemaphoreDescriptor before setting the value of semaphore count and making the semaphore NIL. The user supplies the initial value for the count (see the User's Guide). The P operation on a SEMAPHORE is like a wait on a SIGNAL, but if the count is greater than zero, no wait occurs. Instead the count is decremented and the calling process continues. If the count equals zero, then the process is queued and a TRANSFER to Clock occurs just like a wait for type SIGNAL. The V operation is like a signal operation (on type SIGNAL), except if there is no waiting process, the count is incremented; otherwise the next process in the queue is made ready and the queue is moved forward. There is no operation on type SEMAPHORE similar to procedure waited on type SIGNAL. This satisfies specification clause S2.1.

3.3. Compiler Bugs

As mentioned earlier, most of the previously detected compiler bugs have been corrected in the new version of Modula-2 on the PDP-11. These include a problem with variable names and with parameterless function declarations. A problem with type REAL still exists, however. Two distinct, but perhaps related problems exist. Constants and variables of type REAL appear to be limited
in size to the maximum size of an INTEGER, ie, \(+-32767\). This is apparently an error in the way the compiler sets them up and the way the run-time system converts and does arithmetic. It is possible that an incorrect compiler option was selected when the compiler was built. There are several REAL-handling options for the compiler, including several hardware floating-point processors and a software simulator for floating-point operations.

The second error relates to accuracy of floating-point calculations. They are extremely inaccurate. Again, this may be due to incorrect set-up of the compiler when it was installed. At the present time these problems do not seem critical, but it will eventually be desirable to rebuild the Modula-2 system in order to see if they can be fixed.

Another problem exists, relating to the way Modula-2 restores RT-11 system parameters when it ends. The problem with the clock interrupt vector mentioned in conjunction with the scheduler module is similar. When Modula (the run-time system, not a user program) ends, it does not correctly restore some of the system address correctly, and as a result certain RT-11 programs will crash the system. One such is the RT-11 command 'SHOW'.
'SHOW' alone works correctly, but if 'SHOW ALL' is given as a command, it attempts to check more devices than 'SHOW' and causes RT-11 to crash, requiring a re-boot. Fixing this requires a correction in the Macro routines of the run-time system for Modula-2.
There are many possible ways of thinking about and implementing concurrency in computer programs. Several of these have been discussed briefly in Chapter 2 of this paper. The PreemptiveScheduler module developed during Phase One of the project supplies the basic components of concurrency (starting and stopping concurrent processes and interaction control), but does not provide some of the higher-level structures which have been proposed. Thus far, the system is still a straight Modula-2 programming environment; PreemptiveScheduler is just a standard MODULE which can be used by the programmer. Further development of concurrency techniques will lead further and further away from standard Modula-2 syntax and usage.
In order to allow a student (or other programmer) to work with the higher-level constructs (monitors, etc.), the development of pre-processors has been proposed. These pre-processors would allow extensions to Modula-2's syntax or the use of other syntax during program development, and the pre-processor would convert that syntax into a correct Modula-2 MODULE. This has the advantage to students of allowing them to experiment with some of the other concurrent programming constructs which have been proposed. These constructs often have additional advantages (to students and others) of allowing a more logical way of thinking about concurrency, and of not requiring the programmer to work with low level types and operators (like semaphores). The conversion from high-level constructs to low-level operations would be handled by the pre-processor.

A Level-1 data flow diagram and PSA document (see Appendix D) for the Pre-Processor (Phase Two) follows.

The concept of the monitor, formulated by Per Brinch Hansen and developed by Hoare [1974] and Brinch Hansen [1977a,1977b], is the next step above semaphores as a means of controlling concurrent processes. The monitor itself provides for mutual exclusion of user processes, preventing simultaneous access to critical regions which
are hidden by the monitor. It also provides the programmer with SIGNAL and WAIT operations (similar to, but more complex than, those provided by PreemptiveScheduler) for use in controlling interactions between processes using the monitor. The monitor itself is essentially a program structure similar to a Modula-2 MODULE. Brinch Hansen's Concurrent Pascal language [BrinchHansen, 1977a] has the monitor defined as a data type, allowing the programmer to declare multiple copies of a given monitor. This is not possible in Modula-2, but the basic requirements of the monitor (mutual exclusion and interaction control) can be developed within a Modula-2 module. Ben-Ari [1982] provides algorithms for implementing monitors in terms of semaphore operations.

4.1. Analysis

The monitor, as previously described, requires essentially two things: mutual exclusion for processes using the monitor, and provision for SIGNAL and WAIT operations to allow control over interactions between user processes. The discussion in chapter 5 of Ben-Ari [1982] is recommended reading for information on how a monitor works. Essentially, a monitor is a collection of procedures, usually operating on some local data structure, and only one process at a time can be using a
monitor procedure. When a process is required to wait, it must release control of the monitor so that another process can enter. When a process exits the monitor, it must also release the mutual exclusion control. If a process signals to allow a waiting process to proceed, it must ensure that it exits the monitor or else must go into a wait state.

Two forms of the SIGNAL operation have been described by Ben-Ari [1982] and others. The simpler form requires that a SIGNAL, if one exists, be the last operation before a process exits the monitor. This means that if some process is waiting to be signaled, it will be re-started and the current process will exit; if no process is waiting, the current process will simply release mutual exclusion (waiting processes have priority over processes which have not yet entered the monitor). The more complicated (and more general) version of the SIGNAL can be used anywhere within the monitor, but requires that the signaling process enter a wait state when it signals the waiting process to proceed. Exiting processes must also ensure that any process which has been thus halted be allowed to proceed before any outside process can enter. How to do this will be discussed in detail under the Design section.
An initial consideration was to decide how a monitor should be used within a Modula-2 programming environment. It would be possible to provide a pre-processor to convert programs written in Concurrent Pascal into Modula-2, but to do so would mean that the programmer would no longer be concerned with Modula-2 at all. It seems more reasonable to simply extend Modula-2 to allow monitors to be programmed directly. The first step in the analysis phase was therefore to write a draft of the user manual for the pre-processor, describing how to write a monitor in Modula-2. (The final version of the User's Guide is in Appendix B.)

Essentially, the pre-processor should be expected to take care of the mutual exclusion problem completely; it will no longer be a concern of the programmer. In addition, Modula-2 syntax must be extended to allow use of type 'CONDITION' and the operators 'SIGNAL' and 'WAIT' on that type. The pre-processor will convert these extensions into correct Modula-2 statements which can be compiled.

Since mutual exclusion for monitor processes only applies to those procedures which are exported, the first thing the pre-processor must do is get a list of exported procedures by reading the DEFINITION MODULE.
The programmer will write the DEFINITION MODULE just as for a normal Modula-2 module. The monitor itself will be written exactly like a normal IMPLEMENTATION MODULE except for use of the extensions mentioned above. After finding out which procedures are exported, the preprocessor must add the required mutual exclusion statements, and convert the extensions (CONDITION, SIGNAL, and WAIT) into the correct implementation format. Since concurrency will be provided by use of the Preemp­tiveScheduler module, it will need to be imported, and the pre-processor should also take care of this.

The following Level-2 data flow diagram shows this process. A PSA document explaining the data flow diagram is in Appendix D. It should be noted that this analysis is based upon how a human programmer would convert a monitor into a Modula-2 module, probably using a text editor. This means that a machine executable preprocessor may not work in exactly the same way. This will be discussed further in the following sections.

4.2. Design

4.2.1. Initial Design

The initial design strategy was to implement a program with structure very similar to that portrayed in the
Process 1.1 Level-2 Data Flow Diagram
Module-2 System
data flow diagram. While it would probably be possible
to do the transform from monitor to Modula-2 module in
a single pass (ie, read through the monitor file once
making the required changes and output the IMPLEMENTA­
TION MODULE), it is easier to conceptualize it as a
multi-pass transform, adding the mutual exclusion con­
trols on one pass and converting the CONDITION state­
ments on another pass. Multi-pass transforms require
some means of passing the partially transformed data
from one pass to the next, and this is usually done
using intermediate files. Since a concurrent program­
ming system is available, however, it would be just as
easy, and perhaps more efficient, to make the separate
passes into concurrent processes (instead of sequential
procedures), and have a buffer between them instead of
a file.

One of the ideas used for design is the fact that
Modula-2 is a word-oriented language. That is, key­
words and user-defined names within programs are
separated by a distinctive character set. In the case
of Modula-2, keywords and user names must be formed
from alphabetic characters (a-z,A-Z) or digits (0-9)
and may not include any punctuation, special charac­
ters, or white space. The design for the pre-processor
is based upon this fact in that it uses a parser to
find 'words' and uses these 'words' to form tokens for locating particular keywords needed to convert the input file. The pre-processor is not intended to be a complete Modula-2 compiler and will not attempt to recognize more of Modula-2's syntax than necessary. Only a few of the Modula-2 keywords will be recognized as tokens for the pre-processor and everything else will be copied directly from the input file to the output file without any other action. When keywords are found and their presence is appropriate, the required conversion activity will be triggered.

At this point in the design process a preliminary design was laid out. In reviewing it, it became apparent that some consideration must be given to the algorithm for implementing the monitor transform before the design could be completed. In order to do this, the consumer-producer program using a bounded buffer monitor given by Ben-Ari [1982, pp75ff] was programmed in Modula-2. The monitor was programmed exactly as a programmer will be expected to write such a monitor, and then it was converted into an executable module by hand in order to test the monitor algorithms as they were developed. Development of the algorithm is discussed in the following section. It will be apparent that this has major implications for the final program
4.2.2. Monitor Algorithm

Initially it was thought that using the restricted form of the SIGNAL statement would be easier to implement since the unrestricted form requires declaration of extra variables. Ben-Ari [1982] gives algorithms for both forms [pp86–90], and these are the basis of the following discussion. A table of the algorithms given by Ben-Ari and their possible implementations in Modula-2 is shown on the following page. Each of these will be covered in this discussion.

One consideration in developing the monitor preprocessor is to create as little burden for the programmer as possible. Three keywords have already been added to Modula-2 by the monitor: CONDITION, SIGNAL(COND)[4], and WAIT(COND). It is desired not to have too many extra things which are forbidden to the programmer. Obviously the SEMAPHORE type and its operations, plus the variables needed for mutual exclu-

[4] In the following discussion, the reader must be careful to differentiate between the TYPE SIGNAL exported by module PreemptiveScheduler and the operation SIGNAL(COND) on TYPE CONDITION which is used only within a monitor. The operation signal(SIGNAL) is used on TYPE SIGNAL.
<table>
<thead>
<tr>
<th>Initialization &amp; Notes:</th>
<th>ENTRY:</th>
<th>EXIT:</th>
<th>WAIT:</th>
<th>SIGNAL:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A:</strong> Ben-Ari,1982</td>
<td>SIGNAL required to be last</td>
<td>P(s);</td>
<td>V(s); (* no SIGNAL *)</td>
<td>cc := cc + 1;</td>
</tr>
<tr>
<td></td>
<td>s : semaphore := 1</td>
<td></td>
<td>V(s);</td>
<td>P(c);</td>
</tr>
<tr>
<td></td>
<td>c : semaphore := Ø</td>
<td></td>
<td>P(c);</td>
<td>cc := cc - 1;</td>
</tr>
<tr>
<td></td>
<td>cc : integer := Ø</td>
<td></td>
<td>if cc&gt;0</td>
<td>then V(c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>else V(s);</td>
</tr>
<tr>
<td><strong>B:</strong> first attempt in Modula-2 (after A:)</td>
<td>P(s);</td>
<td>V(s); (* no SIGNAL *)</td>
<td>V(s); wait(c);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s : SEMAPHORE; initsemaphore(s,1);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c : SIGNAL; initsignal(c);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cc := cc + 1;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if cc&gt;0</td>
<td></td>
<td>IF waited(c)</td>
<td>THEN signal(c)</td>
</tr>
<tr>
<td></td>
<td>then V(c) ELSE V(s) END;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C:</strong> Ben-Ari,1982</td>
<td>unrestricted SIGNAL</td>
<td>P(s);</td>
<td>if uc&gt;0</td>
<td>cc := cc + 1;</td>
</tr>
<tr>
<td></td>
<td>s : semaphore := 1</td>
<td></td>
<td>then V(u)</td>
<td>uc := uc + 1;</td>
</tr>
<tr>
<td></td>
<td>uc : integer := Ø</td>
<td></td>
<td>else V(s);</td>
<td>if uc&gt;0</td>
</tr>
<tr>
<td></td>
<td>u : semaphore := Ø</td>
<td></td>
<td></td>
<td>then V(u);</td>
</tr>
<tr>
<td></td>
<td>c : semaphore := Ø</td>
<td></td>
<td>else V(s);</td>
<td>else V(s);</td>
</tr>
<tr>
<td></td>
<td>cc : integer := Ø</td>
<td></td>
<td>P(c);</td>
<td>P(u); end;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cc := cc - 1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>uc := uc - 1;</td>
<td></td>
</tr>
<tr>
<td><strong>D:</strong> Modula-2 (after C:) unrestricted SIGNAL</td>
<td>P(s);</td>
<td>IF u.cc&gt;0</td>
<td>INC(c.cc);</td>
<td>INC(u.cc);</td>
</tr>
<tr>
<td></td>
<td>s : SEMAPHORE: initsemaphore(s,1);</td>
<td>THEN V(u,c)</td>
<td>IF u.cc&gt;0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TYPE CONDITION = RECORD</td>
<td>ELSE V(s) END;</td>
<td>THEN V(u,c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c : SEMAPHORE;</td>
<td>ELSE V(s) END;</td>
<td>ELSE V(s) END;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cc : CARDINAL END;</td>
<td>P(c);</td>
<td>P(c);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>u : CONDITION; initsemaphore(u,c,Ø);</td>
<td>DEC(c.cc);</td>
<td>DEC(u.cc);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>u.cc := Ø;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c : CONDITION; initsemaphore(c,c,Ø);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c.cc := Ø;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PROBLEM:</strong> If an interrupt occurs during execution of the V(s) in WAIT (see arrow), then waited(c) in SIGNAL may incorrectly be FALSE when a SIGNAL is executed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sion will be in this category. In looking at Ben-Ari's algorithm (A: in the Monitor Algorithm Table) for the restricted signal monitor, we see that for each declared CONDITION variable we need a semaphore and a counter. Doing away with the counter is an obvious simplification. It is used only to indicate that some process is waiting for the CONDITION to be signaled. Since there is a function 'waited' for the type 'SIGNAL' exported by the scheduler, it would seem feasible to use type SIGNAL instead of type SEMAPHORE, and do away with the count variable. Such an implementation is shown at B: in the table.

This implementation was actually written into the boundedbuffer monitor and tested. Under the test conditions it works quite nicely, but critical examination reveals that it has a flaw (as pointed out in the table). Since we are using a time-sharing process scheduler, consideration must be given to what happens when the clock interrupt occurs. If an interrupt occurs during the execution of the WAIT (as implemented), it is possible for an immediately following SIGNAL (by another process which enters the monitor) to be 'lost' since it may not know that another process is waiting. The signal(SIGNAL) exported by PreemptiveScheduler does not have a 'memory' the way a
semaphore does (i.e., a V(sem) preceding a P(sem) will cause the P not to wait, while if a signal(SIGNAL) precedes a wait(SIGNAL) a wait will still occur). Thus this implementation of WAIT(COND) is faulty. This points up a problem with concurrent processes: since there is indeterminacy involved in their execution, testing may not reveal significant errors; concurrent programs must be 'proven' in some more or less rigorous manner if at all possible. If proof is not possible, extremely rigorous testing must be done.

Once this problem was discovered, further investigation of the algorithms was done. Eventually the possibility of simply declaring a TYPE CONDITION was recognized, and since the unrestricted SIGNAL only requires one extra variable, its implementation appeared more reasonable. With CONDITION as a type, Ben-Ari's algorithm (C: in the table) translates directly into Modula-2 (D: in the table). Since Ben-Ari's algorithm was proven correct [pp 88-90 of Ben-Ari], and since the implementation is a direct translation, this implementation is correct. Not only that, but it lends itself perfectly to modularization. Both SIGNAL and WAIT can be encapsulated into procedures, and since CONDITION will be a declared type, no conversion is needed at all for it or its operations; the type and procedure declarations
simply need to be put into the code. Monitor entry and exit can also be encapsulated, but this may not be desirable since a procedure call involves more system overhead and less efficiency of execution. This solution was then placed into the test monitor and executed.

There is one possible problem with implementing monitors using semaphores. If a procedure within the monitor calls one of the exported procedures, deadlock will result. (Thanks are due to Dr. Barr for detecting this error.) As a result, programmers must be forbidden to use such design constructs. This should not result in any serious problems as long as it is considered during the design phase. There does not appear to be any way of avoiding this difficulty using Modula-2. Languages which include monitors as direct program constructs may be able to solve the problem since mutual exclusion could be handled by the compiler, and calls from within the monitor could avoid multiple occurrences of the mutual exclusion calls (at entry and exit).

4.2.3. Final Design

With this algorithm, a single pass solution is apparent. It is no longer necessary to convert the condition type and operations. Initialization is
required for variables of type CONDITION, but this will be done in the main portion of the monitor and can be inserted as part of a single sequential operation. The final program design is shown on the following page; the diagram is done using the Yourdan-Constantine system [Yourdan,1979]. A pdsl design document from the SEM system details the design (see Appendix D).

As noted previously, this design does not follow the data flow diagram very closely. The design has module Convert, which is essentially a transform centered design, rather than the 'pipe-line' form shown in the data flow. The analysis was done by looking at what the conversion should consist of without considering how it should be carried out by a computer. The change to the final form of the design was very dependent upon the algorithm used to implement the conversion, and this algorithm is somewhat dependent upon the implementation language (Modula-2). The algorithm could be implemented in some other languages, but its relation to Modula-2 and the PreemptiveScheduler module is very strong.

The pdsl document also reveals some considerations not completely realized in the design diagram. In particular, module Convert has several local procedures, pri-
marily in order to make the algorithm for the parser more understandable through use of modularization. There are also a number of abstract data types and locally important variables declared in the pdsl document which do not appear in the design diagram. It is expected that further refinement of the design will take place during the implementation phase.

4.3. Implementation

4.3.1. Preliminaries

The implementation of the pre-processor obviously required opening several files, preferably without having to continuously prompt the user for file names (as required by module InOut). Additionally, it was not desirable to write local routines to do the job just for the pre-processor. Therefore, the file I/O module FInOut was implemented at this time. This was primarily a conversion job, involving combining InOut and RealInOut, and then eliminating the terminal I/O functions. In order to gain maximum flexibility, module FilePool was used to obtain the number of the next free channel to be opened. The system version of FilePool halted the program if no channel was available, so it had to be modified to return an error indicator, allowing the user program to handle the error. Module InOut
was modified (first by Dr. Barr and later by Phil Rosine) to also use FilePool for channel assignment. (Wirth's version used constants to assign the I/O channels used.) FInOut then had to be tested and InOut retested, and the I/O User Guide was extended to cover the additions. (See the Appendices for the source code, User's Guide, and test programs.)

4.3.2. Implementation and Test Plan

The following plan refers to items defined in the design. The implementation plan is devised so that each module and routine can be tested as it is implemented. Much of the testing can be done by making 'dummy' modules for any pieces not implemented and using the MONITR program itself to test its subparts. Note that some names have been modified from the pdsl document since the period (.) is not allowed in Modula-2 names.

(1) **Program MONITR.** The main program for the preprocessor is relatively simple, and can be implemented immediately. By making 'dummy' modules for the main routine which need to be imported by MONITR, it can be tested, and then used as a test driver for some of the other portions of the program. As a test driver it will have some output
statements added in order to look at the intermediate data (ExpProcList in particular); these will later be removed.

(2) **Monitor Table File.** This is the file containing the statements which have to be added to the monitor code in order to make it a module which can be compiled. It was partially written when the test program (bounded buffer monitor) was done, and will have to be improved as the program implementation develops.

(3) **Test Data.** A test file was created as a part of testing the algorithm for converting the parts of a monitor into Modula-2. This file will be used as the primary test data. Other data may be developed as needed. (See Appendix C).

(4) **MODULE MData.** This module is not defined in the design, but will be a useful tool for the implementation. This will contain the declarations for the constants, the data types, and some global variables. No testing (other than compilation) will be needed.

(5) **MODULE MFileHandler.** This is a LIBRARY according to the pdsl document. It will be implemented as a
Modula-2 module. Some portions of it (as noted in the design) will probably not be implemented as routines since they are trivial calls to routines in FInOut.

(6) **MODULE MLibrary.** This module is not defined in the design, but was implemented due to need for several general purpose routines. One of these, Token, is listed in the pdsl document as a part of ConvertLibrary, but was put here instead.

(7) **MODULE StrLibrary.** The routines included here are a general purpose string (ARRAY OF CHAR) comparison functions and a general purpose string copy routine. These have potentially wider application than the MONITR program and so were encapsulated in a module.

(8) **MODULE MGetDefs.** The main routine defined in the pdsl document is probably not strictly necessary for this module, but it hides the variable ExpList, and thus is a valuable addition. The routines will be implemented in the order they are called: ReadExportList, followed by FindProcedures.
(9) **MODULE ConvertLibrary.** This (LIBRARY) module will be included in Convert. It was defined under pdsl as a LIBRARY simply because the included routines needed to be called by several other routines, and this is not allowed in pdsl.

(10) **MODULE Convert.** For this module, the main routine is necessary, and several of the routines shown in the pdsl document (but not on the design diagram) will be local routines to the main routine. The routines can be implemented in the order they are called by the algorithm, and can be tested by adding 'debug' statements to the algorithm to print out portions of the data as needed. Since at this point the output file will be being produced, most of the testing will be partial test runs of the whole program.

4.3.3. **Implementation**

The implementation process went very smoothly, particularly for the portions of the program which were somewhat over-designed. The main program, and modules MFileHandler, MLibrary, and MGetDefs in particular were implemented very quickly and accurately. The implementation of MConvert was more troublesome, probably because the design was not carried to the level of
detail that some the previous portions were. Testing of the various parts worked much as planned. A module MTest was used to contain the test drivers and output procedures; these were imported on a temporary basis by the main program or it subparts as needed to test the various pieces. Testing was essential to locating certain bugs in the design and implementation, and it was particularly important to use several files for test data. One early version of the whole program worked perfectly with the primary test module Bounded-Buffer, but when the module MConvert (not intended as a monitor, but used as test data anyway) was used, several errors were discovered, including those involving quoted strings and CASE statements.

Several problems with the design were encountered. The most serious of these was a failure to account for quoted strings within an IMPLEMENTATION MODULE when it was handled by MConvert. This was taken care of fairly easily since a slight modification of the CommentHandler procedure (producing procedure QuoteHandler) solved the problem. Since the occurrence of quoted strings is a fairly restricted event, the trapping of strings for handling by QuoteHandler was quite simple.
CASE statements also turned out to be something which had been forgotten. The vertical bar (|) used in them interfered with some of the algorithms used for locating comments and END statements. These were modified without difficulty to solve the problem.

A third problem area was that comments can sometimes occur in locations where it is troublesome to trap their occurrence. The designed algorithms did not find all of them, and locating all comments turned in somewhat of a headache. Consideration was given to restricting the use of comments in some unusual locations within program code, such as within an EXPORT statement or between the keywords PROCEDURE or END and a following procedure name. After some consideration, a means of doing this comment trapping in a clean, simple manner was found and implemented.

The final problem with the design was that a user may try to run an incomplete file through the preprocessor. In order to counter this possibility a large number of end-of-file checks had to be added to both MConvert and MGetDefs, and a routine added to MLibrary to write out an error message, close the files, and halt the program.
A number of minor changes were made to the design, mostly in the data structures used. The way names were stored underwent some evolution during the implementation phase, and the data structure was modified to assist that process. In several cases the parameters passed to procedures were changed, added, or deleted. While these changes are not insignificant, they can be considered normal, and in this case are rather minimal since such changes relate to the implementation language used. Implementing a structured design of this type in a language like COBOL or BASIC which relies heavily upon global variables would necessitate many more changes than were made in this design.

After the implementation was complete, an new design diagram was done in order to document the design. This appears on the following page. The changes to the design are quite apparent when it is compared to the original design. This implementation diagram shows the routines contained within MConvertMain which were part of the design (but were not shown on the original design diagram).
MONITOR
Monitor Pre-Processor

OpenFile (User) 
GetBlockList (Monitor) 
ReadBlockList (Monitor) 
Preprocessor (Monitor) 

MONITOR as implemented

Parameters
1. Err
2. ExpCondList
3. ExpList
4. FlxCode
5. FlxCond
6. statStates
7. ParseToken
8. ADDToken
9. CondList
10. CType
Chapter 5
Phase Three -- Scheduler Trace

5.1. General Discussion

The idea of a trace feature is to allow a programmer to follow the process interleaving either for debugging or educational purposes. In general, it is desired to have some form of record indicating what process is running and what it is doing relative to the scheduler.

Dr. Dhesi recommended that the following would be desirable features of a trace package.

(1) Each change in the state of a process, along with the current program counter of the process should be recorded.
(2) When a change in state is caused by a signal or semaphore operation, some way of identifying which signal/semaphore the wait/signal occurred on (perhaps by giving the absolute memory address associated with that signal/semaphore).

(3) Recording when I/O interrupts occur would be desirable, but this could be tricky or impossible. (An I/O interrupt just interrupts the process for a very short time, and the process continues without the scheduler necessarily gaining control.)

The first two items are not particularly difficult to provide. The I/O interrupts, however, are beyond the control of the PreemptiveScheduler module, and cannot be recorded without extreme difficulty if at all. This will not be attempted.

5.2. Design of the Trace Feature

Tracing these things appears fairly simple to do on the surface, but there are some hidden problems. The major problem is that it is desirable to have the tracing done with minimal impact upon the action of the scheduler. This is important particularly for debugging since the bug may be a result of some side effect of indeterminacy of process execution, and if the trace action changes
the execution too much it may change the result obtained. Unfortunately, most I/O actions take longer than a clock cycle and will definitely interfere with the action of the scheduler. This almost automatically eliminates the simple expedient of outputting the scheduler's actions as a means of tracing.

Instead of trying to record the trace in a file or on the user's terminal, it was eventually decided to record the trace on a local data structure, and only output it when specifically requested by the user. A data structure consisting of an array of records, each record containing the current process ID, the current program counter value, the action that triggered the trace, the address of any variable involved, and the ID of the next process to be executed. An explanation of these items is in the PreemptiveScheduler User's Guide in Appendix A. To allow efficient use of the computer memory and execution time, the trace feature was included in the DEFINITION module for PreemptiveScheduler, but was only implemented in a special version of the IMPLEMENTATION module. A programmer can decide which version to use when the program is linked.
5.2. Implementation of the Trace Feature

No particular problems were encountered in implementing the trace feature. A copy of the PreemptiveScheduler module was made and the trace code was placed in it. This module was given the file name TRACES.MOD with module name PreemptiveScheduler. The DEFINITION module was modified to include the TraceOutput procedure, and a dummy version of TraceOutput was put into the normal scheduler module. This provides the programmer with the ability to have calls to TraceOutput in a program for debugging purposes, and later relink the program with the smaller (normal) version of the scheduler.

5.4. Data Structures

The record needed for storing the data was implemented exactly as designed. Locating the program counter (PC) of an interrupted process required some extra work, however. Dr. Barr provided assistance in locating the code in the Modula-2 run time system (file RTS.MAC on the LSI-11/23) where the processor register and program counter are written to the process stack during a TRANSFER or IOTRANSFER. The address stored in the PROCESS (data type) variable during a TRANSFER (IOTRANSFER) points to a location in the workspace for the process. The PC must be located relative to this location. In
order to do this a record type was declared as follows:

```
STACK = (* locations of registers in the PROCESS stack *)
RECORD
  R1,R0,R5,R3,R4,R2,PC,PS: WORD
END;
```

and a variable 'stack' was declared to be a POINTER TO STACK. When the program counter is required, the address to the process stack is assigned to 'stack', and 'stack^PC' then has the value of the program counter at the time the process was interrupted. This value can then be stored in the PC portion of the record recording the trace.

5.5. Procedure TraceOutput

The procedure TraceOutput writes the trace history out to a file. The filename and channel number are supplied by the calling program. While output is being done, the clock interrupt is turned off to prevent interruption of the trace. The interpretation of the output is explained in the scheduler user's guide. A copy of a test output is included in Appendix C of this paper. This test is a result of inserting a call to TraceOutput as the last statement in the producer-consumer program (the test program for the monitor pre-processor).
Previous sections of this thesis paper were carefully written in the 'third person' style of technical writing. At this point, however, I find myself in the position of needing to discuss some of the more personal facets of software development as I encountered them during this project. Therefore, the paper will make an abrupt stylistic shift to the first person.

6.1. Project Review

In general, the project went quite well. The development effort generally followed the course outlined during the initial analysis, although it ran somewhat behind schedule and did not quite do everything that was originally desired. The scope of the project
(particularly Phase Two) turned out to be generally too large, and had to be pared down as it progressed. The major aim of the project, to set up a Modula-2 system which could be used for teaching concurrent programming was realized, although not in quite the depth originally envisioned.

6.1.1. Phase One

With one exception, all the goals of Phase One were accomplished. The problems previously encountered with compiler bugs took care of themselves due to a new release of the compiler. The missing modules for I/O were developed and tested. A new bug was discovered concerning real numbers, but was not solved due to time constraints. Because of this bug, the math library for real number arithmetic was not done. Some modification of the plan took place since the PreemptiveScheduler module was developed as a part of Phase One rather than Phase Two. The major reason this phase took much longer than planned was because I felt that I needed to do a great deal of background research and reading. This resulted in Chapter 2 of this paper (Concurrency Overview) and the annotated bibliography. In fact, I feel that I could have spent more time doing this reading, and still have a stack of papers I have not had
time to get into. Time constraints again dictated that I not do more than the minimum required research.

6.1.2. Phase Two

Phase Two originally included all of the concurrent processing software needed to make Modula-2 a language suitable for teaching the concepts of concurrency. Part of this, the time-slicing scheduler, was done during Phase One, where it probably should been placed anyway. Phase Two therefore was devoted to developing a monitor pre-processor. I had intended to develop other pre-processors, but there was not enough time to do the job. The monitor pre-processor actually did not take too long. Most of the time went into the analysis and design phases, which is the way things are supposed to work by the latest theory of software development. The monitor pre-processor shows that the idea of using pre-processors to extend Modula-2 is a valid concept, and further work (such as I had intended) is feasible.

6.1.3. Phase Three

No special problems were encountered while adding the trace feature. The process was facilitated by the logical and modular implementation of the scheduler itself. Modification of a modular program which has
been developed using the concepts of cohesion and connectedness was thus shown to be feasible, and (in my experience) easier than with programs which did not have these features.

6.2. Project Development

Two schemes of project development were actually used during the course of this project. Phases One and Three were done using a heuristic methodology, where the project proceeded by means of a series of tests and refinement. Phase Two, on the other hand, was developed in a rigorous manner, following the structured methodologies as developed by DeMarco, Yourdan, Constantine, and others. The methods merged in some areas, since parts of the analysis for Phase One followed the rigorous method, and part of the algorithm development in Phase Two followed the heuristic method. Actually, the two methods complement each other very nicely, and the combination seems to produce good results.

6.2.1. Analysis

The primary analysis method used was the data analysis method described by DeMarco. This method uses Data Flow Diagrams, a Data Dictionary, and Structured
English process algorithms to describe an existing or proposed system. The basis of the method is to analyze the flow of data through the system rather than concentrating upon the processes involved. The theory is that the processes are defined by the data, rather than the data being defined by the processes. For this project I was primarily describing a proposed system since there was no existing system. This is somewhat different from the normal analysis where an existing system is to be converted to a computerized system. The data flow diagrams seem to work very well, however, and enabled me to visualize and describe the system much easier.

For the data dictionary and process descriptions I used the PSL/PSA program. This generally worked very well, although some parts of PSL do not exactly match the methodology described by DeMarco. Using PSA allowed me to cross check my data analysis and helped to ensure correctness.

During Phase Two I tried the technique of writing the user's manual before designing or implementing the program. This seemed to work very well, and I intend to do it as much as possible in the future. It not only gets part of the documentation out of the way early
(saving much grief after all else is done), but also serves to clarify the requirements and specifications, particularly with regard to the user interface. I considered the original version of the user's manual to be a rough draft, but by having it ready-made, all I had to do was correct the manual when I changed the program in any way that affected how it appeared to the user. In the case of the monitor pre-processor, these changes consisted mainly of adding some error messages to the manual. This is a very effective and efficient technique.

6.2.2. Design

The design process in particular was a mixture of heuristics and formal techniques. Although there were many cross-overs between the two, I will discuss them each briefly.

6.2.2.1. Heuristic Design — As previously mentioned, Phase One was done largely by means of heuristic design. This was particularly true of the development of the scheduler. The I/O package was largely a process of re-writing the supplied modules, or converting them to fit a new use (i.e., INTEGER vs REAL number I/O). The scheduler, although based upon
Wirth's 'Processes' module, was largely uncharted ground, and in addition, I had no experience upon which to base a formal analysis and design. Dr. Barr and Dr. Dhesi supplied me with ideas and explanations which gave some foundation to the problem analysis, but since the design had to very implementation dependent, much of it had to be done by trial and error. Working from the 'Sched' module previously written by Dr. Barr, I had to make changes, test them, interpret the results, and make further changes until I ended up with a working solution. Design and implementation became parts of the same process. At some points it became clear that the design could be made to be more cohesive and less connected by revising the structure of the scheduler module and the procedures within it. I never did make a formal design of the module. Although I tried on at least one occasion, the process simply did not work for this phase of the project, partly due to my lack of knowledge and partly due to the inherent nature of the module.

6.2.2.2. Formal Design — I used the formal design techniques as developed by Yourdan and Constantine [Yourdan, 1979] as the primary design technique for Phase Two of the project. This technique is based upon a graphic style of structure charting (as opposed
to flow charting) using the concepts of maximizing cohesion and minimizing connectedness as a means of guiding the development of the design. Cohesion refers to how strongly the parts of module are related to each other. Connectedness refers to the linkage between separate modules. ('Module' as used by Yourdan and Constantine refers to a program part, not to a Modula-2 MODULE.) Using this method helps produce a correct, easy to understand program. I personally found that the graphic technique was very useful. I made some eight or ten design charts before I got one that satisfied me. Although I did find some minor errors in the design (during implementation), they were very easy to correct due to the cohesiveness and minimal connectedness of the design. As an adjunct to the design charts, I used the Program Design Structure Language (pdsl) of the System Encyclopedia Manager (SEM). (See the section on 'Tools' for more discussion of SEM.)

6.2.3. Implementation and Testing

Implementation and testing was generally a rather easy matter. I found that once I began to understand Modula-2, it was an excellent implementation language. The separate compilation units and the automatic
linkage features lend themselves to fast development. These features made the heuristic development during Phase One work smoothly, and the implementation and testing portions of Phase Two went much faster than expected. Because of the modular design of Modula-2 syntax, implementation and testing plans follow naturally from structured design. I did feel that Modula-2 was somewhat hard to get used to at first, partly due to the lack of standardized I/O, partly due to the stringent type checking, and partly due to the way our compiler handles errors, but once I got used to it I found that I liked it very much.

6.2.4. Tools

The three major tools used during project development were the Modula-2 compiler, linker, and run-time-system, the ISDOS analysis and design tools, and the local computer network. Each of these will be discussed briefly.

6.2.4.1. Modula-2 System — The Modula-2 compiler, linker, and run-time-system which I used has some limitations. The compiler and linker sometimes seem to be a bit slow, but I eventually got used to that. A worse problem is that errors are listed only in the
*.lst file produced by the compiler, so if compile
time errors occur, the programmer has to look through
this file to see what the errors were. This is some-
times painful, particularly at 300 baud. I developed
the habit of using teco (rt-11 editor) to look for the
errors (which are flagged with '****'). The linker is
one of the nicest features about modula-2, since it
knows what to link without need for the programmer to
write out a long command line, although its default
search routine may be overridden to allow linkage of
non-standard modules for test purposes. The only
thing wrong with the run-time-system is that it does
not correctly reset the interrupt vectors after run-
ing a program which uses the iotransfer call, and
this required me to add some code to the scheduler
(one of the reasons for phase one's heuristic design).

6.2.4.2. ISDOS Tools -- I made extensive use of
the ISDOS tools PSA/PSL and sem. These have been dis-
cussed in previous sections. I found that PSA/PSL was
generally useful, mostly as a means of cross-checking
myself. This project was not large enough to really
require all of the features of PSA, which I suspect
really becomes effective only for very large software
development projects. PSL seems to work very nicely
with deMarco's data flow diagrams, and assists the
analysis phase a great deal.

The SEM system runs a subset of PSA/PSL, but I did not use it for this purpose. I used SEM primarily for the design phases of the project, using the pdsl language. pdsl was not really designed for use with the Yourdan-Constantine method of design, although it can be made to work. It was useful as a tool to document the design chart developed during Phase Two, but did not seem to be useful as a primary design tool. Part of this may be personal, since I find that graphic tools generally work better for me. I also found that pdsl did not work at all for the design of concurrent programs. I tried to use it during Phase One, but without success. pdsl seems to be oriented toward implementation languages like FORTRAN which rely heavily upon global data. When information hiding and modularization is attempted it sometimes causes troubles. Part of the difficulty is undoubtably the fact that we have not been able to find out exactly what design technique was intended to be used with pdsl. During Phase Two I developed a logical view of pdsl that finally began to work well, and perhaps if I went back to Phase One I could now apply it successfully. The concept behind the ISDOS software is excellent, but I feel that it needs some further development.
6.2.4.3. **Local Network** — Our local system is configured with a number of terminals and computers (a VAX-11/750 and several LSI-11/23's) connected to a central minicomputer (LSI-11/23) which allows links between computers and terminals. Several LSI's and the VAX are also linked with an ethernet. Since Modula-2 runs only on the LSI machines, I had to do most of my work on them. However, they are slower, and the editor available (TECO) is not a full-screen editor. I eventually decided to do my development work (ie, editing) on the VAX (under UNIX) and transfer the files to and from the LSI using the ethernet link. This worked quite well, and I feel that it increased my productivity a great deal.

6.3. **Conclusions and Lessons Learned**

The overall result of this project for me personally has been two-fold: I have gained knowledge about concurrency and concurrent systems, and I have gained confidence in the development tools and techniques learned over the last few years. Particularly important is the feeling that some of the design and analysis techniques are becoming 'second-nature'. This is especially true of the ideas of cohesiveness and connectedness. I find that I am giving these concepts emphasis in my
programming without having to consciously consider them, and when I run into difficulties, these are first things I look at. Modularization, based upon these principles, is something I strive for even in languages where it is not a featured tool (as it is in Modula-2). Even during Phase One, where the design and implementation were not done formally, I used these ideas.

One technique which is receiving more attention in some of the literature on program design and implementation is the idea of using the compiler to control errors, particularly errors arising from the use of subroutine and function libraries. I found that programming in Modula-2 lead me into using this technique, unconsciously at first and later intentionally. The basic idea is to use data and program structures (particularly subrange data types) to force the compiler to check whether parameters passed to a routine are correct. The strict type checking of Modula-2 is really what allows this to be done. It would be extremely difficult in a language like C which does very little type checking. An example is the file number parameter in module FInOut, which is declared as the subrange type [1..15]. The compiler will require that any call to the routines in FInOut have this parameter within the correct range, thus helping to prevent runtime errors.
I also find that while the tools and techniques are valuable, they are not the most important part of software development. The aim of software engineering is to produce quality software, software that is correct, complete, and easy to work with. The techniques, and the principles upon which they are based, are merely a means, not an end. This is emphasized in this project by the way the two design techniques, the heuristic method and the formal method, worked together to produce the overall result. There is always a temptation in computer science to try to work by rules, forgetting that analysis and programming is really a craft (an artistic craft) and not a science with rigid laws. The tools and techniques assist in the work, but they cannot produce quality by themselves. They must be used, but used judiciously, and the rules must be relaxed where necessary. This of course requires practice and experience, which are hard to come by, and I feel that this thesis project has supplied me with some of that practical experience.
Appendix A

Appendix A -- Source Code

The source code consists of two parts for each Module: the DEFINITION MODULE and the IMPLEMENTATION MODULE. They will occur in that order for each set of code.
DEFINITION MODULE InOut;  (*N.Wirth 19.2.80*)
(* modified by Philip E. Rosine
17 nov 83

...modification was done to make this module
compatible with the definition for InOut
given in [Wirth,1983]. *)

(* General Notes:
Several of these operations are refer to any character
less than or equal to a blank; this is denoted as <=" "
in the comments. Since the system uses ASCII, characters
<=" " include all control characters, tab, carriage
return, and line feed, as well as spaces. *)

FROM SYSTEM IMPORT WORD;
EXPORT QUALIFIED
  EOL, Done, termCH,
  OpenInput, OpenOutput, CloseInput, CloseOutput,
  Read, ReadString, ReadInt, ReadCard,
  Write, WriteLn, WriteString, WriteInt, WriteCard,
  WriteOct, WriteHex;

CONST EOL = 15C;  (* this is a carriage return *)

VAR
  Done : BOOLEAN;
  (* Done is used to indicate the success or failure of
  a procedure call to InOut. The exact meaning is
  dependent upon which call is made... see the notes for
  each procedure. *)
  termCH : CHAR;
  (* termCH will contain the last character read by any of
  the read operations; ie, the character which terminated
  the read, not the last character in data read. This
  character is NOT pushed back on the input stream, and if
  it is needed by the next data item, the calling program
  must add it to the data itself. The initial value of
  termCH is not specified. *)

PROCEDURE OpenInput(defext:ARRAY OF CHAR);
  (* request a file name and open file "in".
  Done := "file was successfully opened".
  If open, subsequent input is read from this file.
  If name does not have an extension, add extension
defext. Default device is DK:, default filename
is IN. If <ESC> is input, TTY: will become the
PROCEDURE OpenOutput(defext:ARRAY OF CHAR);
  (* request a file name and open output file "out".
  Done := "file was successfully opened".
  If open, subsequent output is written on this file.
  If name does not have an extension, add extension
defext. Default device is DK:, default filename
  is OUT. If.<ESC> is input, TTY: will become the
  output. *)

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 in.eof *)

PROCEDURE ReadString(VAR s:ARRAY OF CHAR);
  (* read string, ie, sequence of characters not containing
  control characters, tabs, carriage return, or linefeed.
  Input is terminated by any character < " "; this
  character is assigned to termCH. DEL is used for
  backspacing when input from terminal. The last
  character of the string s is OC. If more characters
  are input than the size of s would allow for, they
  will be lost. Done is TRUE if the number of
  characters read is >0. *)

PROCEDURE ReadInt(VAR x: INTEGER);
  (*skip blanks and control characters; if a sequence of
digits (possibly preceded by a sign) follows, read
it as a decimal integer. Done indicates whether a number
was read, termCH gives the last character read, i.e. the
one following the sequence of digits. No tests for
overflow are made. DEL is used to backspace for terminal
input. Input is terminated by any character <= " ". *)

PROCEDURE ReadCard(VAR x: CARDINAL);
  (* read string and convert to cardinal. Syntax:
cardinal= digit{digit}.
  Leading Blanks are ignored. Done := "cardinal was read".
  DEL is used to backspace for terminal input. Input is
  terminated by any character <= " ". For both ReadCard
  and ReadInt, if the input data includes non-numeric data
  before the termCH (<=" "), the numeric data up to that
  point will be converted and placed in the parameter
  variable, and the following data (up to termCH) will
be lost. Done will be FALSE in this case, although the data in x may be valid. Some data from the input stream will have been lost. *)

PROCEDURE Write(ch: CHAR);

PROCEDURE WriteLn; (* terminate line with carriage return followed by linefeed *)

PROCEDURE WriteString(s: ARRAY OF CHAR);
(* The string s is assumed to be any string of characters up to either the maximum size of the ARRAY s or until the first null (OC) is found. A quoted string constant meets these standards. The compiler will not allow quoted strings to contain characters < " ". Use Write with an octal or hex constant parameter to output control characters, tabs, or form control characters. *)

PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
(* write integer x with (at most) n characters on file "out". If n is greater than the number of digits needed, blanks are added preceding the number. If the number requires more than n characters, the field will be filled with "*". *)

PROCEDURE WriteCard(x, n: CARDINAL);

PROCEDURE WriteOct(w, n: CARDINAL);
(* n must be >=6 *)

PROCEDURE WriteHex(w, n: CARDINAL);
(* n must be >=4 *)

END InOut.
IMPLEMENTATION MODULE InOut; (*$T- N.Wirth 27.2.80*)
(* modified by Philip E. Rosine
17 nov 83
modification was done to make this module compatible with the definition for InOut
given in [Wirth,1983]. *)

IMPORT SYSTEM, TTIO, Files, FilePool, FileNames, Streams;

CONST CR = 15C; LF = 12C; ESC = 33C; DEL = 177C;
NumLength = 6;
VAR InFromTTY, OutOnTTY: BOOLEAN;
in, out: Streams.STREAM;
InFno,OutFno: CARDINAL;

PROCEDURE FormatFilename(VAR outname: ARRAY OF CHAR;
inname: Files.FileName);
(* This routine takes a system format file name and adds punctuation to make it readable. *)
VAR i, j : CARDINAL;
BEGIN
j 0;
FOR i:=0 TO 11 DO
outname[i] := inname[i];
INC(j);
CASE i OF
2 : outname[j] := ";"; INC(j)
8 : outname[j] := "."; INC(j)
END
outname[12] := 0C
END FormatFilename;

PROCEDURE OpenInput(defext: ARRAY OF CHAR);
(* request a file name and open file "in".
Done := "file was successfully opened".
If open, subsequent input is read from this file.
If name does not have an extension, add extension defext. Default device is DK:, default filename
is IN. If <ESC> is input, TTY: will become the input. *)
VAR
ok: INTEGER;
ch: CHAR;
InFN, defname: Files.FileName;
filename : ARRAY [0..15] OF CHAR;
BEGIN
REPEAT

defname := "DK IN ";
defname[9] := defext[0];
defname[10] := defext[1];
WriteString("input = ");
FileNames.ReadFileName(lnFN, defname); TTIO.Read(ch);
IF ch = ESC THEN
  InFromTTY := TRUE; ok := 0
ELSE
  FilePool.GetFileNameNumber(lnFno);
  IF lnFno#0 THEN
    Files.Lookup(lnFno, InFN, ok);
    IF ok >= 0 THEN
      Streams.Connect(in, lnFno, FALSE); InFromTTY := FALSE;
      Done := TRUE
    END
  END;
  IF (lnFno=0) OR (ok<0) THEN
    Done := FALSE;
    WriteLn; WriteString("error in opening input file ");
    FormatFilename(filename, InFN);
    WriteString(filename);
    IF lnFno=0 THEN
      WriteString("; no channel available");WriteLn
    ELSE
      WriteString("; errorcode = ");
      WriteInt(ok,3); WriteLn;
    END
  END;
  TTIO.Write(CR); TTIO.Write(LF);
UNTIL (ok >= 0) OR (lnFno=0);
END OpenInput;

PROCEDURE OpenOutput(defext:ARRAY OF CHAR);
  (* request a file name and open output file "out".
     Done := "file was successfully opened".
     If open, subsequent output is written on this file.
     If name does not have an extension, add extension defext. Default device is DK:, default filename is OUT. If <ESC> is input, TTY: will become the output. *)
VAR
  ok: INTEGER;
  ch: CHAR;
  OutFN, newOutFN, defname: Files.FileName;
  filename :  ARRAY [0..15] OF CHAR;
BEGIN
  REPEAT
    defname := "DK OUT ";

defname[9] := defext[0];
defname[10] := defext[1];
WriteString("output = ");
FileNames.ReadFileName(OutFN, defname); TTIO.Read(ch);
IF ch = ESC THEN
  OutOnTTY := TRUE; ok := 0
ELSE
  FilePool.GetFileNumber(OutFno);
  IF OutFno#0 THEN
    Files.Lookup(OutFno, OutFN, ok);
    IF ok = -2 THEN (* create a new file *)
      Files.Create(OutFno, OutFN, ok);
      IF ok < 0 THEN
        WriteLn;
        WriteString("error in creating output file ")
      END
    ELSEIF ok >= 0 THEN (* file exists, rename _____Q___ *)
      Files.Release(OutFno);
      Files.Rename(OutFno, newOutFN, OutFN, ok);
      IF ok >= 0 THEN (* recreate the output file *)
        Files.Create(OutFno, OutFN, ok);
        IF ok < 0 THEN
          WriteLn;
          WriteString("error in recreating output file ")
        END
      ELSE
        WriteLn;
        WriteString("error in renaming output file ")
      END
    ELSE
      WriteLn;
      WriteString("error in file lookup for ")
    END
  END;
ELSE
  WriteLn;
  WriteString("error in file lookup for ")
END
END;
IF (ok >= 0) & (OutFno#0) THEN
  Done := TRUE;
  Streams.Connect(out, OutFno, FALSE); OutOnTTY := FALSE
ELSE
  Done := FALSE;
  FormatFilename(filename, OutFN);
  IF (OutFno=0) THEN WriteLn;
    WriteString("error in opening output file ");
  END;
  WriteString(filename);
  IF OutFno=0 THEN
    WriteString("; no channel available");
  WriteLn
ELSE

WriteString("; errorcode = ");
WriteInt(ok,3); WriteLn;
END
END
END ;
TTIO.Write(CR); TTIO.Write(LF);
UNTIL (ok >= 0) OR (OutFno=0);
END OpenOutput;

PROCEDURE CloseInput;
(* closes input file; returns input to terminal *)
BEGIN
IF NOT InFromTTY THEN
   Streams.Disconnect(in, TRUE);
   Files.Release(InFno);
   InFromTTY := TRUE
END;
END CloseInput;

PROCEDURE CloseOutput;
(* closes output file; returns output to terminal *)
BEGIN
IF NOT OutOnTTY THEN
   Streams.EndWrite(out);
   Streams.Disconnect(out, TRUE);
   Files.Release(OutFno);
   OutOnTTY := TRUE
END;
END CloseOutput;

PROCEDURE Read(VAR ch: CHAR);
(* Done := NOT in.eof *)
BEGIN
IF InFromTTY THEN TTIO.Read(ch); TTIO.Write(ch);
   CASE ch OF
      CR : TTIO.Write(LF)
   | LF : TTIO.Write(CR)
   END
ELSE Streams.ReadChar(in, ch)
END;
IF ch = OC THEN Done := FALSE ELSE Done := TRUE END
END Read;

PROCEDURE delete;
(* delete a character entered from a terminal —
   this procedure will erase the character from a CRT, but
   will not remove it from the input data area *)
BEGIN
IF InFromTTY THEN
TTIO.Write(10C);TTIO.Write(40C);TTIO.Write(10C)
END
END delete;

PROCEDURE ReadString(VAR s:ARRAY OF CHAR);
(* read string, i.e., sequence of characters not containing
control characters. Input is terminated by any character < " "; this character is assigned to termCH.
DEL is used for backspacing when input from terminal.
The last character of the string s is 0C. *)
VAR
  ch: CHAR;
  CHcount: CARDINAL;
BEGIN
  CHcount := 0;
  Read(ch);
  WHILE (ch >= " ") DO
    IF ch = DEL THEN
      IF CHcount > 0 THEN delete; DEC(CHcount)
      ELSIF InFromTTY THEN Write(7C) END
    ELSE
      IF CHcount < HIGH(s) THEN
        s[CHcount] := ch; INC(CHcount)
      ELSIF InFromTTY THEN delete; Write(7C)
      END
    END;
  Read(ch)
END;
  s[CHcount] := 0C; (* end of string *)
  termCH := ch;
  IF CHcount = 0 THEN
    Done := FALSE
  ELSE
    Done := TRUE
  END;
END ReadString;

PROCEDURE ReadInt(VAR x: INTEGER);
(* skip blanks and control characters; if a sequence of
digits (possibly preceded by a sign) follows, read it as a decimal integer. Done indicates whether a number
was read, termCH gives the last character read, i.e. the one following the sequence of digits. 0C signals the end
of the input stream. No tests for overflow are made.
DEL is used to backspace for terminal input. Input is
terminated by any character <= " ". *)
VAR
  ch: CHAR;
  neg: BOOLEAN;
  xo, CHcount, counter: INTEGER;
incoming: ARRAY [0..numLength] OF CHAR;

BEGIN
  chcount := 0;
  read(ch);
  while (ch <= " ") & (ch # OC) & (ch # CR) & (ch # LF) DO
    read(ch) END;
  while ch > " " DO
    if ch = DEL THEN
      if chcount > 0 THEN delete; dec(chcount)
      elseif infromtty THEN write(7C) END
    else
      if chcount <= numLength THEN
        incoming[chcount] := ch; inc(chcount)
      elseif infromtty THEN delete; write(7C)
      end;
  end;
  read(ch)
END;

termch := ch;
if chcount = 0 THEN
  done := FALSE
else
  done := true;
  neg := false;
  counter := 0;
  if incoming[0] = "-" THEN neg := true; counter := 1
  elseif incoming[0] = "+" THEN counter := 1 END;
  if (counter < chcount) and
      ((incoming[counter] >= 0")
    & (incoming[counter] <= "9")
  ) THEN
    x0 := val(integer,ord(incoming[counter]) - ord('0'));
    inc(counter);
    while (counter < chcount) and done DO
      if (incoming[counter] >= "0")
    & (incoming[counter] <= "9") THEN
        x0 := 10*x0 + val(integer,ord(incoming[counter])
      - ord('0'));
        inc(counter)
      else
        done := false
      end
    end;
    if neg THEN x := -x0 ELSE x := x0 END
  else
    done := false
  end
END
PROCEDURE readint(var x: cardinal);
(* read string and convert to cardinal. Syntax:
  cardinal=digit{digit}.
Leading Blanks are ignored. Done := "cardinal was read". termCH contains the last character read. DEL is used to
backspace for terminal input. Input is terminated by any
ccharacter <= ".
*)

VAR
  ch: CHAR;
  x0, CHcount, counter: CARDINAL;
  incoming: ARRAY [0..NumLength] OF CHAR;
BEGIN
  CHcount := 0;
  Read(ch);
  WHILE (ch <= ".") & (ch # OC) & (ch # CR) & (ch # LF) DO
    Read(ch) END;
  WHILE ch > "." DO
    IF ch = DEL THEN
      IF CHcount > 0 THEN delete; DEC(CHcount) END
    ELSE
      IF CHcount <= NumLength THEN
        incoming[CHcount] := ch; INC(CHcount)
      ELSIF InFromTTY THEN delete; Write(7C) END
      END;
    Read(ch)
  END;
  termCH := ch;
  IF CHcount = 0 THEN
    Done := FALSE
  ELSE
    counter := 0;
    Done := TRUE;
    IF (incoming[counter] >= "0") & (incoming[counter] <= "9") THEN
      x0 := VAL(CARDINAL, ORD(incoming[counter]) - ORD(\'0\'));
      INC(counter);
      WHILE (counter < CHcount) AND Done DO
        IF (incoming[counter] >= "0")
          & (incoming[counter] <= "9") THEN
            x0 := 10*x0 + VAL(CARDINAL, ORD(incoming[counter])
                      - ORD(\'0\'));
        INC(counter)
      ELSE
        Done := FALSE
      END
    END
    END;
  x := x0
ELSE
  Done := FALSE
END
PROCEDURE Write(ch: CHAR);
BEGIN
  IF OutOnTTY THEN TTI0.Write(ch)
  ELSE Streams.WriteByte(out, ch)
END Write;

PROCEDURE WriteLn;
BEGIN
  Write(CR); Write(LF);
END WriteLn;

PROCEDURE WriteString(s: ARRAY OF CHAR);
VAR i: CARDINAL;
BEGIN i := 0;
  WHILE (i <= HIGH(s)) & (s[i] # OC) DO
    Write(s[i]); INC(i)
END;
END WriteString;

PROCEDURE WriteNumber(NumString: ARRAY OF CHAR;
  StringSize, FieldSize: CARDINAL);
  (* Writes a number, filling the field with '*' if it is
     too small or padding with blanks if too large. *)
BEGIN
  (** NumString contains a number in reverse order **) 
  (** StringSize>=1 **) 
  IF FieldSize < StringSize THEN 
    FOR StringSize:=1 TO FieldSize DO Write('*') END 
  ELSE 
    WHILE FieldSize > StringSize DO 
      Write(' '); DEC(FieldSize) END;
  (** StringSize>=1 :: required upon entry **) 
  REPEAT 
    DEC(StringSize); Write(NumString[StringSize])
  UNTIL StringSize = 0
END;
END WriteNumber;

PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
  (* write integer x with (at most) n characters on file
    "out". If n is greater than the number of digits needed, 
    blanks are added preceeding the number. If the number 
    requires more than n characters, the field will be filled 
    with '*' . *)
VAR i, x0: CARDINAL;
a: ARRAY [0..NumLength] OF CHAR;
BEGIN i := 0; x0 := ABS(x);
REPEAT
  a[i] := CHR(x0 MOD 10 + ORD('0'));
  x0 := x0 DIV 10;
  INC(i);
UNTIL x0 = 0;
(** i>=i **) IF x < 0 THEN a[i] := "-"; INC(i) END ; 
WriteNumber(a,i,n)
END WriteInt;

PROCEDURE WriteCard(x, n: CARDINAL);
VAR i: CARDINAL;
a: ARRAY [0..NumLength] OF CHAR;
BEGIN i := 0;
REPEAT
  a[i] := CHR((x MOD 10) + ORD('0'));
  x := x DIV 10; INC(i)
UNTIL x = 0;
(** i>=i **) WriteNumber(a,i,n)
END WriteCard;

PROCEDURE WriteOct(w, n: CARDINAL);
VAR i: CARDINAL;
a: ARRAY [0..NumLength] OF CHAR;
BEGIN i := 0;
REPEAT
  a[i] := CHR((w MOD 8) + ORD('0'));
  w := w DIV 8; INC(i)
UNTIL i = 5;
IF w = 0 THEN a[i] := CHR(ORD('0'))
ELSE a[i] := CHR(ORD('1')) END ; 
INC(i);
WriteNumber(a,i,n)
END WriteOct;

PROCEDURE WriteHex(w, n: CARDINAL);
PROCEDURE HexDig(d: CARDINAL);
BEGIN d := d MOD 16;
  IF d < 10 THEN d := d+ORD('0') ELSE d := d-10+ORD('A') END;
  Write(CHR(d))
END HexDig;
BEGIN
  IF n < 4 THEN
    REPEAT Write("*"); DEC(n) UNTIL n<=0;
  ELSE
    WHILE n > 4 DO Write(' '); DEC(n) END;
  END;
BEGIN

HexDig(w DIV 1000H); HexDig(w DIV 100H);
HexDig(w DIV 10H); HexDig(w)
END;
END WriteHex;

BEGIN

InFromTTY := TRUE;
OutOnTTY := TRUE;
Done := FALSE;
END InOut.
DEFINITION MODULE RealInOut;

(* as defined by Wirth, 1983, pp104-5

entered by Philip E. Rosine
25 Nov 83 *)

EXPORT QUALIFIED ReadReal, WriteReal, WriteRealOct, Done;

VAR Done: BOOLEAN;

PROCEDURE ReadReal(VAR x: REAL);

(* Read REAL number x according to syntax:

["+" I  "-"Jdigit{digit} ["."digit{digit}] ["E"["+"I "-"Jdigit{digit}]

Done := "a number was read".
At most 7 digits are significant, leading zeroes not counting. Maximum exponent is 38. Input terminates with a blank or any control character. DEL is used for backspacing.*)

PROCEDURE WriteRealCx: REAL; n: CARDINAL);

(* Write x using n characters. If fewer than n characters are needed, leading blanks are inserted. *)

PROCEDURE WriteRealOct(x: REAL);

(* Write x in octal form — two octal words are written showing the memory representation of the real x — fieldwidth is 13 *)

END RealInOut.
IMPLEMENTATION MODULE RealInOut;

(* as defined by Wirth, 1983, ppl04-5

implemented by Philip E. Rosine
25 Nov 83 *)

IMPORT InOut;
FROM InOut IMPORT ReadString, WriteOct, Write;
FROM SYSTEM IMPORT WORD, ADDRESS, ADR, TSIZE;

CONST stringSize = 80;

PROCEDURE ReadReal(VAR x: REAL);

(* Read REAL number x according to syntax:
    ["+"|"-"|"[0-9].[0-9]
    
    Done := "a number was read".
    At most 7 digits are significant, leading zeroes not counting. Maximum exponent is 38. Input terminates with a blank or any control character. DEL is used for backspacing. *)

VAR i: CARDINAL;
string: ARRAY [0..stringSize] OF CHAR;

PROCEDURE GetMantissa;

(* Get the mantissa of the number from the string starting at character position i. *)

VAR neg: BOOLEAN;
x1, divisor: REAL;
BEGIN
    neg := FALSE;
    IF string[i]="-" THEN neg := TRUE; INC(i)
    ELSIF string[i]="+" THEN INC(i)
    END;
    IF (string[i] >= "0") & (string[i] <= "9") THEN
        (* get integer portion of number *)
        x := FLOAT(VAL(INTEGER,ORD(string[i]) - ORD('0')));
        INC(i);
        WHILE (string[i] >= "0") & (string[i] <= "9") DO
            x := 10.0*x + FLOAT(VAL(INTEGER,ORD(string[i]) - ORD('0')));
            INC(i)
        END;
    IF string[i]="." THEN (* get decimal portion *)
        INC(i);
        divisor := 10.0;
        WHILE (string[i] >= "0") & (string[i] <= "9") DO
xl := FLOAT(VAL(INTEGER,ORD(string[i]) - ORD('0')));
x := x + xl/divisor;
INC(i);
divisor := divisor*10.0
END
END;
IF neg THEN x := -x END (* negate if required *)
END GetMantissa;

PROCEDURE GetExponent;
(* Get the exponent of the number from the string
starting at character position i. *)
VAR times, j: CARDINAL;
negexp: BOOLEAN;
exponent, multiplier: REAL;
BEGIN
INC(i);
negexp := FALSE;
IF string[i]="-" THEN
negexp := TRUE;
INC(i)
ELSIF string[i]="+" THEN INC(i)
END;
BEGIN (* get size of exponent *)
j := 0;
WHILE (string[i] >= "0") & (string[i] <= "9") DO
j := j*10 + VAL(CARDINAL,ORD(string[i]) - ORD('0'));
INC(i)
END;
IF j <= 38 THEN
IF j > 0 THEN
exponent := 1.0;
IF negexp THEN multiplier := 0.1
ELSE multiplier := 10.0 END;
FOR times:=1 TO j DO
exponent := exponent*multiplier END;
x := x * exponent
END
ELSE Done := FALSE
END
END GetExponent;

BEGIN (* ReadReal *)
ReadString(string);
IF InOut.Done THEN (* a string was read in *)
i := 0;
(* find first character of real *)
WHILE (string[i]>OC) & (string[i]=" ") DO INC(i) END;
IF string[i] = OC THEN Done := FALSE (* end of input *)
ELSE
x := 0.0;
Done := TRUE;
GetMantissa;
IF string[i]="E" THEN (* check for exponent portion *)
  GetExponent
END
ELSE (* error from InOut.ReadString *)
  Done := FALSE
END
END ReadReal;

PROCEDURE WriteReal(x: REAL; n: CARDINAL);
(* Write x using n characters. If fewer than n
characters are needed, leading blanks are inserted. *)
VAR
  string,reverse: ARRAY [0..stringSize] OF CHAR;
x0: INTEGER;
i,j,m: CARDINAL;
BEGIN
  IF n > stringSize THEN n := stringSize + 1 END;
i := 0;
  IF x < 0.0 THEN
    string[i] := "-";
x := -x;
    INC(i)
  END;
x0 := TRUNC(x);
j := 0;
  (* get the integer portion of the number in reverse order -*-
WHILE (x0 > 0) & (INTEGER(j+i-2) <= INTEGER(stringSize)) DO
  reverse[j] := CHR(CARDINAL(x0 MOD 10) + ORD('0'));
x0 := x0 DIV 10;
  INC(j)
END;
(* (j+i) == the number of cells with valid data in 2 ARRAYS *)
(* cells 0..(j+i-2) will have valid data in one ARRAY *)
IF (j+i) > n THEN (* field size error *)
  FOR j:=1 TO n DO Write('*') END
ELSE
  IF j > 0 THEN
    (* put the reverse integer into the output
    string in correct order *)
    FOR m:=(j-1) TO 0 BY -1 DO
      string[i] := reverse[m];
      INC(i)
    END
  ELSE
    string[i] := '0';
  END
END WriteReal;
INC(i)
END;
IF i < n THEN (* put in decimal point *)
  string[i] := ".";
  INC(i)
END;
x := x - FLOAT(TRUNC(x));
WHILE (i < n) & (x > 0.0) DO (* get fraction *)
  x := x * 10.0;
  string[i] := CHR(CARDINAL(TRUNC(x)) + ORD("0"));
  x := x - FLOAT(TRUNC(x));
  INC(i)
END;
(* right justify in field *)
WHILE i < n DO Write(" "); DEC(n) END;
DEC(i);
FOR j:=0 TO i DO Write(string[j]) END
END;  
END WriteReal;

PROCEDURE WriteRealOct(x: REAL);
(* Write x in octal form — two octal words are written
  showing the memory representation of the real x *)
VAR
  adrx : ADDRESS;
BEGIN
  adrx := ADR(x);
  WriteOct(CARDINAL(adrx^),6); Write(" ");
  adrx := adrx + TSIZE(WORD);
  WriteOct(CARDINAL(adrx^),6)
END WriteRealOct;

BEGIN (*RealInOut*)
  Done := FALSE
END RealInOut.
MODULE FilePool; (* LG 05.06.80 *)

(* for RT-11 operating system *)
(* modified by Phil Rosine
Mar 84

The original version of this module halted if there were no available file numbers. The module only uses file numbers 1..15; since number 0 is not normally available through GetFileNumber, the module has been changed to return 0 as an errorcode indicating that no number are available. It is the users responsibility to handle this. The modules InOut and FInOut, which use FilePool, will not open a file if there are no numbers available, and use their own return codes to so inform the caller.
*)

EXPORT QUALIFIED GetFileNumber;

PROCEDURE GetFileNumber(VAR filenum: CARDINAL);
(* get file number of free channel *)
(* returns 0 if none available, otherwise a CARDINAL on the range 1..15 *)

END FilePool.
IMPLEMENTATION MODULE FilePool; (* LG 05.06.80 *)

(* modified by Phil Rosine
Mar 84

The original version of this module halted if there were no available file numbers. The module only uses file numbers 1..15; since number 0 is not normally available through GetFileNumber, the module has been changed to return 0 as an errorcode indicating that no number are available. It is the users responsibility to handle this. The modules InOut and FInOut, which use FilePool, will not open a file if there are no numbers available, and use their own return codes to so inform the caller. *)

FROM SYSTEM IMPORT RT11CALL;
FROM Files IMPORT Errcode;

VAR lastnum : CARDINAL;

PROCEDURE GetFileNumber(VAR filenum: CARDINAL);
VAR num : CARDINAL;
err : BOOLEAN;
BEGIN
num := lastnum;
LOOP
num := num MOD 15 + 1; (* channel 0 is not touched *)
IF num = lastnum THEN filenum := 0; EXIT END;
RT11CALL(374B,num,err);
IF err AND (Errcode = 0C) THEN (* free channel found *)
filenum := num;
lastnum := num;
EXIT;
END;
END GetFileNumber;

BEGIN
lastnum := 15;
END FilePool.
DEFINITION MODULE FinOut;
(* modified by Philip E. Rosine
27 feb 84

This is a modification of InOut in order to allow opening of multiple files for input or output. *)

IMPORT Files;

EXPORT QUALIFIED
  FEOL, FDone, FtermCH, ChanNum,
  FOpenInput, FOpenOutput, FCloseInput, FCloseOutput,
  FRead, FReadString, FReadInt, FReadCard,
  FWrite, FWriteLn, FWriteString, FWriteInt, FWriteCard,
  FWriteOct, FWriteHex, FReadReal, FWriteReal, FWriteRealOct;

CONST FEOL = 15C; (* this is a carriage return *)

TYPE ChanNum = [1..15]; (* channel numbers *)

VAR
  FDone : BOOLEAN;
  (* FDone is used to indicate the success or failure of a procedure call to InOut. The exact meaning is dependent upon which call is made... see the notes for each procedure. *)
  FtermCH : CHAR;
  (* FtermCH will contain the last character read by any of the read operations; ie, the character which terminated the read, not the last character in data read. This character is NOT pushed back on the input stream, and if it is needed by the next data item, the calling program must add it to the data itself. The initial value of FtermCH is not specified. *)

(* It is more or less assumed that the programmer can remember whether a channel is open for input or output, and read/write appropriately. However, in the case of a read operation to a write channel, FtermCH is assigned OC (end-of-file) and FDone will be FALSE. For a write operation to a read channel, FDone will be assigned FALSE. These can be checked if necessary. The critical error checks for the programmer are to insure that no more than 15 files are open at any given time. This includes any files opened by calls to InOut.OpenInput or InOut.OpenOutput. FDone will be returned as FALSE if this happens, but the system will not print an error message. Error messages for this case are the programmers responsibility. The same result will be obtained from trying to open file which is already open (which could happen due to excessive implementation hiding in IMPORTed modules. *)
PROCEDURE FOpenInput(InFno:ChanNum;InFN:Files.FileName);
(* open file InFN for input on channel InFno. 
FDone := "file was successfully opened". 
If open, subsequent input is read from this file. *)

PROCEDURE FOpenOutput(OutFno:ChanNum;OutFN:Files.FileName);
(* open output file OutFN on channel OutFno. 
FDone := "file was successfully opened". 
If open, subsequent output is written on this file. *)

PROCEDURE FCloselnput(InFno:ChanNum);
(* closes input file *)

PROCEDURE FCloseOutput(OutFno:ChanNum);
(* closes output file *)

PROCEDURE FRead(Fno:ChanNum;VAR ch: CHAR);
(* FDone := NOT in.eof *)

PROCEDURE FReadString(Fno:ChanNum;VAR s:ARRAY OF CHAR);
(* read string, ie, sequence of characters not containing 
control characters, tabs, carriage return, or linefeed. 
Input is terminated by any character < " "; this 
character is assigned to FtermCH. The last character 
of the string s is 0C. If more characters are input 
than the size of s would allow for, they will be lost. 
FDone is TRUE if the number of characters read is >0. *)

PROCEDURE FReadInt(Fno:ChanNum;VAR x: INTEGER);
(*skip blanks and control characters; if a sequence of 
digits (possibly preceded by a sign) follows, read 
it as a decimal integer. FDone indicates whether a number 
was read, termCH gives the last character read, i.e. the 
one following the sequence of digits. No tests for 
overflow are made. Input is terminated by any character 
<=" ". *)

PROCEDURE FReadCard(Fno:ChanNum;VAR x: CARDINAL);
(* read string and convert to cardinal. Syntax: 
cardinal=digit(digit). 
Leading Blanks are ignored. FDone := "cardinal was read". 
Input is terminated by any character <=" ". For both 
ReadCard and ReadInt, if the input data includes non­ 
numeric data before the FtermCH (<=" "), the numeric data 
up to that point will be converted and placed in the 
parameter variable, and the following data (up to
FtermCH) will be lost. FDone will be FALSE in this case, although the data in x may be valid. Some data from the input stream will have been lost. *)

PROCEDURE FWrite(Fno:ChanNum;ch: CHAR);

PROCEDURE FWriteLn(Fno:ChanNum);
(* terminate line with carriage return followed by linefeed *)

PROCEDURE FWriteString(Fno:ChanNum;s: ARRAY OF CHAR);
(* The string s is assumed to be any string of characters up to either the maximum size of the ARRAY s or until the first null (0C) is found. A quoted string constant meets these standards. The compiler will not allow quoted strings to contain characters "". Use Write with an octal or hex contant parameter to output control characters, tabs, or form control characters. *)

PROCEDURE FWriteInt(Fno:ChanNum;x: INTEGER; n: CARDINAL);
(* write integer x with (at most) n characters on file "out". If n is greater than the number of digits needed, blanks are added preceding the number. If the number requires more than n characters, the field will be filled with "\*". *)

PROCEDURE FWriteCard(Fno:ChanNum;x, n: CARDINAL);
(* n must be >=6 *)

PROCEDURE FWriteOct(Fno:ChanNum;w, n: CARDINAL);
(* n must be >=4 *)

PROCEDURE FWriteHex(Fno:ChanNum;w, n: CARDINAL);

PROCEDURE FReadReal(Fno:ChanNum;VAR x: REAL);
(* Read REAL number x according to syntax:
["+"|"-"]digit(digit)["."digit(digit)]['E'"[+"|"-"]digit[digit]]
FDone := "a number was read".
At most 7 digits are significant, leading zeroes not counting. Maximum exponent is 38. Input terminates with a blank or any control character. *)

PROCEDURE FWriteReal(Fno:ChanNum;x: REAL; n: CARDINAL);
(* Write x using n characters. If fewer than n characters are needed, leading blanks are inserted. *)

PROCEDURE FWriteRealOct(Fno:ChanNum;x: REAL);
(* Write x in octal form -- two octal words are written showing the memory representation of the real x -- fieldwidth is 13 *)
END FInOut.
IMPLEMENTATION MODULE FlnOut;

(* modified by Philip E. Rosine 
27 feb 84 
This is a modification of InOut in order to allow 
opening of multiple files for input or output. *)

IMPORT SYSTEM, Files, FilePool, FileNames, Streams;

CONST CR = 15C; LF = 12C; ESC = 33C; DEL = 177C;
NumLength = 6;
stringSize = 80;

TYPE
  FileStatus = (Closed,Input,Output);
  Stream = RECORD
    Fnumber:CARDINAL;
    Fstream:Streams.STREAM;
    status:FileStatus
  END;

VAR
  Channel: ARRAY ChanNum OF Stream;
i: ChanNum;

PROCEDURE FOpenlnput(InFno:ChanNum;InFN:Files.FileName);
VAR
  ok: INTEGER;
  ch: CHAR;
  filename : ARRAY [0..15] OF CHAR;
BEGIN
  WITH Channel[InFno] DO
    FilePool.GetFileNumber(Fnumber);
    IF Fnumber#0 THEN
      Files.Lookup(Fnumber,InFN,ok);
      IF ok >= 0 THEN
        Streams.Connect(Fstream,Fnumber,FALSE);
        status := Input;
        FDone := TRUE
      ELSE
        FDone := FALSE;
      END
    ELSE
      FDone := FALSE;
    END
  END;
FDone := TRUE;
END

PROCEDURE FOpenOutput(OutFno:ChanNum;OutFN:Files.FileName);
VAR
  ok: INTEGER;
  ch: CHAR;
  newOutFN: Files.FileName;
  filename : ARRAY [0..15] OF CHAR;
BEGIN
  WITH Channel[OutFno] DO
    FilePool.GetFileNumber(Fnumber);
    IF Fnumber#0 THEN
      Files.Lookup(Fnumber, OutFN, ok);
      IF ok = -2 THEN (* create a new file *)
        Files.Create(Fnumber, OutFN, ok);
      ELSIF ok >= 0 THEN (* file exists, rename _____Q__ *)
        Files.Release(Fnumber);
        Files.Rename(Fnumber,newOutFN,OutFN,ok);
        IF ok >= 0 THEN (* recreate the output file *)
          Files.Create(Fnumber, OutFN, ok);
      END;
    END;
    IF ok >= 0 THEN
      FDone := TRUE;
      Streams.Connect(Fstream,Fnumber,TRUE);
      status := Output;
    ELSE
      FDone := FALSE;
    END
  ELSE
    FDone := FALSE;
  END
END FOpenOutput;

PROCEDURE FCloseInput(InFno:ChanNum);
BEGIN
  WITH Channel[InFno] DO
    IF status = Input THEN
      Streams.Disconnect(Fstream,TRUE);
      FDone := FALSE;
      status := Closed
    ELSE
      FDone := FALSE
    END
  END;
END FCloseInput;

PROCEDURE FCloseOutput(OutFno:ChanNum);
BEGIN
  WITH Channel[OutFno] DO
    IF status = Output THEN
Streams.EndWrite(Fstream);
Streams.Disconnect(Fstream,TRUE);
FDone := TRUE;
status := Closed
ELSE
FDone := FALSE
END
END
END FCloseOutput;

PROCEDURE FRead(Fno:ChanNum;VAR ch: CHAR);
BEGIN
IF Channel[Fno].status = Input THEN
Streams.ReadChar(Channel[Fno].Fstream,ch);
FDone := TRUE
ELSE
FtermCH := 0C;
FDone := FALSE
END
END FRead;

PROCEDURE FReadString(Fno:ChanNum;VAR s:ARRAY OF CHAR);
VAR
ch: CHAR;
CHcount: CARDINAL;
BEGIN
CHcount := 0;
FRead(Fno,ch);
WHILE (ch >= " ") DO
IF CHcount < HIGH(s) THEN
s[CHcount] := ch; INC(CHcount)
END;
FRead(Fno,ch)
END;
s[CHcount] := 0C; (* end of string *)
FtermCH := ch;
IF CHcount = 0 THEN
FDone := FALSE
ELSE
FDone := TRUE
END;
END FReadString;

PROCEDURE FReadInt(Fno:ChanNum;VAR x: INTEGER);
VAR
ch: CHAR;
neg: BOOLEAN;
x0, CHcount, counter: INTEGER;
ingoing: ARRAY [0..NumLength] OF CHAR;
BEGIN
CHcount := 0;
FRead(Fno,ch);
WHILE (ch <= " ") & (ch # OC) & (ch # CR) & (ch # LF) DO
  FRead(Fno,ch) END;
WHILE ch > " 
  IF CHcount <= NumLength THEN
    incoming[CHcount] := ch; INC(CHcount)
  END;
  FRead(Fno,ch)
END;
FtermCH := ch;
IF CHcount = 0 THEN
  FDone := FALSE
ELSE
  FDone := TRUE;
  neg := FALSE;
  counter := 0;
  IF incoming[0] = "-" THEN neg := TRUE; counter := 1
  ELSIF incoming[0] = "+
  IF (counter < CHcount) AND
    ((incoming[counter] >= "0")
   & (incoming[counter] <= "9"); THEN
    x0 := VAL(INTEGER,ORD(incoming[counter]) - ORD('0'));
   INC(counter);
   WHILE (counter < CHcount) AND FDone DO
    IF (incoming[counter] >= "0")
       & (incoming[counter] <= "9"); THEN
      x0 := 10*x0 + VAL(INTEGER,ORD(incoming[counter])
   INC(counter)
   ELSE
      FDone := FALSE
END
END;
IF neg THEN x := -x0 ELSE x := x0 END
ELSE
  FDone := FALSE
END
END;
END FReadInt;

PROCEDURE FReadCard(Fno:ChanNum;VAR x: CARDINAL);
VAR
  ch: CHAR;
  x0, CHcount, counter: CARDINAL;
  incoming: ARRAY [0..NumLength] OF CHAR;
BEGIN
  CHcount := 0;
  FRead(Fno,ch);
WHILE (ch <= " ") & (ch ≠ 0C) & (ch ≠ CR) & (ch ≠ LF) DO
  FRead(Fno,ch) END;
WHILE ch > " 
  DO IF CHcount <= NumLength THEN
    incoming[CHcount] := ch; INC(CHcount)
  END;
  FRead(Fno,ch)
END;
FtermCH := ch;
IF CHcount = 0 THEN
  FDone := FALSE
ELSE
  counter := 0;
  FDone := TRUE;
  IF (incoming[counter] >= "0") & (incoming[counter] <= "9") THEN
    x0 := VAL(CARDINAL,ORD(incoming[counter]) - ORD('0'));
    INC(counter);
    WHILE (counter < CHcount) AND FDone DO
      IF (incoming[counter] >= "0") & (incoming[counter] <= "9") THEN
        x0 := 10*x0 + VAL(CARDINAL,ORD(incoming[counter]) - ORD('0'));
      INC(counter)
    ELSE
      FDone := FALSE
    END
END
x := x0
ELSE
  FDone := FALSE
END
END FReadCard;

PROCEDURE FWrite(Fno:ChanNum;ch: CHAR);
BEGIN
  IF Channel[Fno].status = Output THEN
    Streams.WriteByte(Channel[Fno].Fstream,ch);
  FDone := TRUE
ELSE
  FDone := FALSE
END
END FWrite;

PROCEDURE FWriteLn(Fno:ChanNum);
BEGIN
  FWrite(Fno,CR); FWrite(Fno,LF);
END FWriteLn;
PROCEDURE FWriteString(Fno:ChanNum; s: ARRAY OF CHAR);
VAR i: CARDINAL;
BEGIN i := 0;
  WHILE (i <= HIGH(s)) & (s[i] # 0C) DO
    FWrite(Fno, s[i]); INC(i)
  END;
END FWriteString;

PROCEDURE FWriteNumber(Fno:ChanNum;
  NumString: ARRAY OF CHAR;
  StringSize, FieldSize: CARDINAL);
BEGIN
  (** NumString contains a number in reverse order **)
  (** StringSize>=1 **) IF FieldSize < StringSize THEN
  FOR StringSize:=1 TO FieldSize DO FWrite(Fno, ' ') END
  ELSE
    WHILE FieldSize > StringSize DO
      FWrite(Fno, ' '); DEC(FieldSize) END;
    (** ==> FieldSize=StringSize **) (** StringSize>=1 :: required upon entry **)
    REPEAT
      DEC(StringSize); FWrite(Fno, NumString[StringSize])
      UNTIL StringSize = 0
    END;
END FWriteNumber;

PROCEDURE FWritelnt(Fno:ChanNum; x: INTEGER; n: CARDINAL);
VAR i, xO: CARDINAL;
a: ARRAY [0..NumLength] OF CHAR;
BEGIN i := 0; xO := ABS(x);
  REPEAT
    a[i] := CHR(xO MOD 10 + ORD('O'));
    xO := xO DIV 10;
    INC(i);
  UNTIL xO = 0;
    (** i>=1 **) IF x < 0 THEN a[i] := "-"; INC(i) END;
    FWriteln(Fno, a, i, n)
END FWritelnt;

PROCEDURE FWriteCard(Fno:ChanNum; x, n: CARDINAL);
VAR i: CARDINAL;
a: ARRAY [0..NumLength] OF CHAR;
BEGIN i := 0;
  REPEAT
    a[i] := CHR((x MOD 10) + ORD('O'));
    x := x DIV 10;
    INC(i);
  UNTIL x = 0;
    (** i>=1 **)
FWriteNumber(Fno,a,i,n)
END FWriteCard;

PROCEDURE FWriteOct(Fno:ChanNum;w, n: CARDINAL);
VAR i: CARDINAL;
a: ARRAY [0..NumLength] OF CHAR;
BEGIN i := 0;
REPEAT
  a[i] := CHR((w MOD 8) + ORD('0'));
  w := w DIV 8; INC(i)
UNTIL i = 5;
IF w = 0 THEN a[i] := CHR(ORD('0'))
  ELSE a[i] := CHR(ORD('1')) END ;
INC(i);
FWriteNumber(Fno,a,i,n)
END FWriteOct;

PROCEDURE FWriteHex(Fno:ChanNum;w, n: CARDINAL);
PROCEDURE HexDig(d: CARDINAL);
BEGIN d := d MOD 16;
  IF d < 10 THEN d := d+ORD('0') ELSE d := d-10+ORD('A') END 
  FWrite(Fno,CHR(d))
END HexDig;
BEGIN
  IF n < 4 THEN
    REPEAT FWrite(Fno,'*'); DEC(n) UNTIL n<=0;
  ELSE
    WHILE n > 4 DO FWrite(Fno,' '); DEC(n) END;
    HexDig(w DIV 1000H); HexDig(w DIV 100H);
    HexDig(w DIV 10H); HexDig(w)
  END;
END FWriteHex;

PROCEDURE FReadReal(Fno:ChanNum;VAR x: REAL);
VAR i: CARDINAL;
string: ARRAY [0..stringSize] OF CHAR;

PROCEDURE GetMantissa;
  (* Get the mantissa of the number from the string
  starting at character position i. *)
VAR neg: BOOLEAN;
x1, divisor: REAL;
BEGIN
  neg := FALSE;
  IF string[i]="-" THEN neg := TRUE; INC(i)
  ELSIF string[i]="+" THEN INC(i)
  END;
  IF (string[i] >= "0") & (string[i] <= "9") THEN
    (* get integer portion of number *)
x := FLOAT(INTEGER,ORD(string[i]) - ORD('0'));  
INC(i);
WHILE (string[i] >= "0") & (string[i] <= "9") DO
  x := 10.0*x + FLOAT(INTEGER,ORD(string[i]) - ORD('0'));  
  INC(i)
END
END;

IF string[i]="." THEN (* get decimal portion *)
  INC(i);
  divisor := 10.0;
  WHILE (string[i] >= "0") & (string[i] <= "9") DO
    x := x + x/divisor;
    INC(i);
    divisor := divisor*10.0
  END
END;

IF neg THEN x := -x END (* negate if required *)
END GetMantissa;

PROCEDURE GetExponent;
  (* Get the exponent of the number from the string starting at character position i. *)
VAR times, j: CARDINAL;
  negexp: BOOLEAN;
  exponent, multiplier: REAL;
BEGIN
  INC(i);
  negexp := FALSE;
  IF string[i]="-" THEN
    negexp := TRUE;
  INC(i)
  ELIF string[i]="+" THEN INC(i)
  END;
  j := 0;
  (* get size of exponent *)
  WHILE (string[i] >= "0") & (string[i] <= "9") DO
    j := j*10 + VAL(CARDINAL,ORD(string[i]) - ORD('0'));  
    INC(i)
  END;
  IF j <= 38 THEN
    IF j > 0 THEN
      exponent := 1.0;
      IF negexp THEN multiplier := 0.1
        ELSE multiplier := 10.0 END;
      FOR times:=1 TO j DO
        exponent := exponent*multiplier END;
      x := x * exponent
    END
END
ELSE FDone := FALSE
END
END GetExponent;

BEGIN (* FReadReal *)
  FReadString(Fno,string);
  IF FDone THEN (* a string was read in *)
    i := 0;
    (* find first character of real *)
    WHILE (string[i]>0C) & (string[i]<=" ") DO INC(i) END;
    IF string[i] = 0C THEN FDone := FALSE (* end of input *)
    ELSE
      x := 0.0;
      FDone := TRUE;
      GetMantissa;
      IF string[i]="E" THEN (* check for exponent portion *)
        GetExponent
      END
      ELSE (* error from ReadString *)
        FDone := FALSE
      END
    END (* FReadReal *)
END

PROCEDURE FWriteReal(Fno:ChanNum;x: REAL; n: CARDINAL);
VAR
  string,reverse: ARRAY [0..StringSize] OF CHAR;
  xO: INTEGER;
  i,j,m: CARDINAL;
BEGIN
  IF n > StringSize THEN n := StringSize + 1 END;
  i := 0;
  IF x < 0.0 THEN
    String[i] := "-";
    x := -x;
    INC(i)
  END;
  xO := TRUNC(x);
  j := 0;
  (* get the integer portion of the number in reverse order *)
  WHILE (xO > 0) & (INTEGER(j+i-2) <= INTEGER(StringSize)) DO
    reverse[j] := CHR(CARDINAL(xO MOD 10) + ORD('0'));
    xO := xO DIV 10;
    INC(j)
  END;
  (* (j+i) == the number of cells with valid data in 2 ARRAYs *)
  (* cells 0..(j+i-2) will have valid data in one ARRAY *)
  IF (j+i) > n THEN (* field size error *)
    FOR j:=1 TO n DO FWrite(Fno,'*') END
ELSE
  IF j > 0 THEN
    (* put the reverse integer into the output string in correct order *)
    FOR m:=(j-1) TO 0 BY -1 DO
      string[i] := reverse[m];
      INC(i)
    END
  ELSE
    string[i] := '0';
    INC(i)
  END;
  IF i < n THEN (* put in decimal point *)
    string[i] := '.';
    INC(i)
  END;
  x := x - FLOAT(TRUNC(x));
  WHILE (i < n) & (x > 0.0) DO (* get fraction *)
    x := x * 10.0;
    string[i] := CHR(CARDINAL(TRUNC(x))) + ORD('0');
    x := x - FLOAT(TRUNC(x));
    INC(i)
  END;
  (* right justify in field *)
  WHILE i < n DO FWrite(Fno,' '); DEC(n) END;
  DEC(i);
  FOR j:=0 TO i DO FWrite(Fno,string[j]) END
END;
END FWriteReal;

PROCEDURE FWriteRealOct(Fno:ChanNum;x: REAL);
VAR
  adr : SYSTEM.ADDRESS;
BEGIN
  adr := SYSTEM.ADR(x);
  FWriteOct(Fno,CARDINAL(adr~),6); FWrite(Fno," ");
  adr := adr + SYSTEM.TSIZE(SYSTEM.WORD);
  FWriteOct(Fno,CARDINAL(adr~),6)
END FWriteRealOct;

BEGIN
  FDone := FALSE;
  FOR i:=1 TO 15 DO Channel[i].status:=Closed END
END FInOut.
DEFINITION MODULE IOControl;

(* written by Phil Rosine
Jan 84

This module provides the means to control access to I/O
functions when running concurrent processes. *)

EXPORT QUALIFIED AcquireIO, ReleaseIO;

PROCEDURE AcquireIO;
(* This procedure guarantees the user process absolute
access to the I/O functions so protected, provided that
all user processes call AcquireIO prior to using I/O. It
cannot guarantee against improper use. *)

PROCEDURE ReleaseIO;
(* This releases I/O which was previously acquired. *)

(* USAGE EXAMPLE ::

P1: P2:

... 
AcquireIO; AcquireIO;
WriteString(\"test P1\" ); WriteCard(P2\^ID);
WriteLn; WriteLn;
ReleaseIO; ReleaseIO;
...

*)

END IOControl.
IMPLEMENTATION MODULE IOControl;

(* written by Phil Rosine
Jan 84

This module controls use of I/O device by means of
a BOOLEAN variable INUSE and a signal inuse. Only
the first caller of AcquireIO gets control; others
will have to wait until the first user calls ReleaseIO,
and then the next process in the wait queue will get
control. *)

FROM PreemptiveScheduler IMPORT SEMAPHORE, P, V, initsemaphore;

VAR
  inuse: SEMAPHORE;

PROCEDURE AcquireIO;
BEGIN
  P(inuse)
END AcquireIO;

PROCEDURE ReleaseIO;
BEGIN
  V(inuse)
END ReleaseIO;

BEGIN
  initsemaphore(inuse,1)
END IOControl.
DEFINITION MODULE PreemptiveScheduler;

(* modified and rewritten by Phil Rosine  
30 Dec 83

This is a scheduler module which implements pre-emptive scheduling of processes. Processes are placed in the ready queue by calling start. The scheduling process is initiated by the first call to start. Since the scheduler uses the system clock interrupt vector, the final action of the main program must be a call to HaltScheduler. This restores the original clock interrupt vector and ensures that the operating system will not crash when the program ends. See the notes with procedures start, HaltScheduler, and HaltProcess.

Wirth ["Programming in Modula-2", 1982, p129ff] discusses the use of implementation module priority levels to implement monitor constructs. As the following discussion demonstrates, this is not precisely true, since setting priorities either will not ensure exclusive access to a module, or will prevent concurrent processing. The implementation module for the scheduler runs at priority level [6], and the hardware clock interrupt also has priority [6]. This means that the clock interrupt will not interrupt calls to the scheduler, and that process interleaving will not take place while any user process is running at level [6] or [7]. This allows monitor effects to be simulated by modules with priority [6] or [7], but at the cost of losing concurrency. A call to such a module will execute until it either finishes with a normal return, until it calls a lower priority module (for instance, and I/O routine), or until it calls wait(signal) or P(semaphore). Since a call to a lower (than [6]) priority module will cause scheduling to resume, the monitor 'effect' is lost, and another process could call the 'monitor' routine successfully even though there is already a process executing it. For this reason, the use of priority levels with this scheduler must be done with extreme care.

If all processes are ever forced into a wait state (this should never happen), the scheduler will force the main program to restart. The function StatusOfScheduler will always return an INTEGER code indicating the current status of the scheduler.

The signal and wait operations on type SIGNAL are primarily synchronization primitives for use with the scheduler. The P and V operations for type SEMAPHORE are
primarily for mutual exclusion in order to control access to critical resources.

(*) modified 12 Apr 84 — per

Added a trace feature consisting of the procedure TraceOutput.

There are two IMPLEMENTATION MODULES: one with the trace feature implemented and one with a dummy trace. See the note under TraceOutput.

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*###########################*
*# #*
*# Copyright 1984. Philip E. Rosine #*
*# #*
*# The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and all parts of the Monitor Pre-processor program. #*
*# #*
*# The type SIGNAL is used with the signal and wait operations to control process synchronization.
*# type SIGNAL is used with the signal and wait operations to control process synchronization.
*# type SIGNAL is used with the signal and wait operations to control process synchronization.
*# type SIGNAL is used with the signal and wait operations to control process synchronization.
* # * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

FROM SYSTEM IMPORT ADDRESS;
FROM FInOut IMPORT ChanNum;
FROM Files IMPORT FileName;

EXPORT QUALIFIED wait, waited, signal, start, SIGNAL, ProcessID,
initsignal, HaltProcess, HaltScheduler,
StatusOfScheduler, P, V, SEMAPHORE, initsemaphore, TraceOutput;

TYPE SIGNAL; (* type SIGNAL is used with the signal and wait operations to control process synchronization.
signal and wait are the operations as defined by Hoare for monitors, but processes under Modula are not monitors. *)

SEMAPHORE; (* type SEMAPHORE is a full-range CARDINAL valued semaphore as defined by Dykstra. *)

(* both SIGNAL and SEMAPHORE must be initialized using initsignal or initsemaphore prior to use. If this is not done, no errors will be generated, but results are indeterminate. *)

PROCEDURE StatusOfScheduler() : INTEGER;
(* indicates current state of the scheduling process: 
Errors in the use of the scheduler will not generate error messages, but will be reflected by StatusOfScheduler, and can be checked and error messages generated by the user program. *)

PROCEDURE ProcessID(): CARDINAL;
(* This function returns the unique process ID number assigned by the scheduler when a process is started. The main program will always have ID 0, and other processes will have consecutively higher numbers in the order they are started. The value returned is undefined prior to the first start. *)

PROCEDURE wait( VAR s: SIGNAL );
(* force process to wait for signal(s) to continue; if the scheduler clock is not running (ie, no concurrent processes have been `start`ed), wait has no action — it becomes effectively a NO-OP; note that the scheduler is halted when all concurrent processes have been halted *)

PROCEDURE waited( VAR s: SIGNAL ):BOOLEAN;
(* TRUE if a process is waiting on this SIGNAL *)

PROCEDURE signal( VAR s: SIGNAL );
(* mark process waiting s as ready to proceed; if no process waiting, then nothing is done *)

PROCEDURE start( p:PROC; a:ADDRESS; n:CARDINAL );
(* start a process p with workspace a of length n *)
(* `starting` a process involves placing it in the ready queue. The first call to start initiates the clock interrupt process which interleaves the processes in the ready queue. A started process should only be exited through a call to HaltProcess. If started processes are `infinite`, the main program must call HaltScheduler prior to ending. *)

PROCEDURE initsignal( VAR s: SIGNAL );
(* initialization of a SIGNAL *)

PROCEDURE HaltScheduler;
(* Halts the scheduling process and restores the clock interrupt vector... required at the end of any program *)
which uses the scheduler. *)

PROCEDURE HaltProcess;
    (* removes a started process from the ready queue —
    if all 'start'ed processes are stopped by calls to
    HaltProcess, an automatic call to HaltScheduler is done
    by the scheduler and the main program restarted at
    whatever point it last was running. *)

PROCEDURE P(VAR s: SEMAPHORE);
    (* tests to see if some process is already using s; if not
    P acquires access to the region guarded by s, otherwise
    pauses until s is released. *)

PROCEDURE V(VAR s: SEMAPHORE);
    (* releases s *)

PROCEDURE initsemaphore(VAR s: SEMAPHORE; initvalue: CARDINAL);
    (* initializes s to initvalue; initvalue determines how the
    SEMAPHORE will work. If it is a positive value n, then
    n concurrent process will be allowed to access whatever
    region s is protecting. If it is 0, then nothing will be
    allowed access to the protected region until a V operation
    has been performed. This will usually be used when
    synchronization primitive is needed. *)

PROCEDURE TraceOutput(Tfile: ChanNum; filename: FileName);
    (* This procedure causes the history of the Preemptive-
    Scheduler (for aproximately the last 5 seconds) to
    be written out to file 'filename'. The trace
    history has the form:
    ProcessID/PC/Action/Variable/NextID
    where ProcessID is the CARDINAL number assigned to the
    current process by the scheduler. PC is the address of
    the next instruction (ie, the Program Counter). PC only
    has validity if the Action code is C, otherwise it will
    be zero. Action is a code as follows:
    I = initial entry of process into the process chain
    C = change of process due to time-slicing
    H = halted by a call to HaltScheduler
    X = halted due to normal procedure end
    W = caused to wait on a SIGNAL
    S = released by a signal on a SIGNAL
    P = caused to wait on a SEMAPHORE
    V = released by a V on a SEMAPHORE
    Variable is the address of the procedure being started
    as a process if Action is 'I', and is the address of the
    SIGNAL or SEMAPHORE if Action is 'W', 'S', 'P', or 'V';
    in all other cases it will be 0 and has no meaning. NextID
is the number of the next process to be executed by the scheduler (all of these shifts are recorded when the scheduler is trying to switch processes — ProcessID is the one being switched from and NextID is the one being switched to: they may be the same). NextID will not have any meaning if Action is 'S' or 'V', and will be set to 65535 in these cases.

(* NOTE:
There are two implementation modules for this definition: the file PREEMP.* has a dummy trace feature and procedure TraceOutput has no action. The file TRACES.* has the actual trace feature implemented. To use the package, compile normally, and link with the /q switch if the trace feature is required. When prompted for PreemptiveScheduler, enter 'traces' in order to link in the trace package.

*)

END PreemptiveScheduler.
IMPLEMENTATION MODULE PreemptiveScheduler [6];
(*$T-,SS-* *)

(* original version by N. Wirth
 modified by Dr. John Barr
 modified and partially re-written by Phil Rosine
 Jan 84

This module implements a pre-emptive scheduler. The Clock
routine will interrupt all processes and start the next
process in the ready queue if possible.
*)

(* modified 12 Apr 84 — per

Added a trace feature consisting of the procedure
TraceOutput.

There are two IMPLEMENTATION MODULES: one with the trace
feature implemented and one with a dummy trace. This
version has a dummy trace.
*)

FROM SYSTEM IMPORT WORD, PROCESS, ADDRESS, ADR, SIZE, TSIZE,
NEWPROCESS, TRANSFER, IOTRANSFER, LISTEN, SYSRESET;
FROM Storage IMPORT ALLOCATE;
FROM FInOut IMPORT ChanNum;
FROM Files IMPORT FileName;

CONS T
ClockTrapAddress = 100B;
ClockVectorAddress = ClockTrapAddress + 2;
PSWAddress = 177776B;
  (* the following define the return codes for the
   Scheduler Status — see notes with StatusOfScheduler *)
RunState = -1;
InitState = 0;
ProcHaltState = 1;
SchedHaltState = 2;
InfiniteWaitState = 3;

TYPE
  SIGNAL = POINTER TO ProcessDescriptor;
  ProcessDescriptor =
    RECORD
      pp: PROCESS;
      proc: PROC; (* procedure to run *)
      ID: CARDINAL; (* PCount when process started *)
      ready: BOOLEAN; (* indicates the run status of the process *)
      next: SIGNAL; (* ties the descriptors in the ring *)
      queue: SIGNAL (* ties processes waiting for the same signal *)
    END ;
  SEMAPHORE = POINTER TO SemaphoreDescriptor;
  SemaphoreDescriptor = RECORD
    count: CARDINAL;
    semqueue: SIGNAL
  END;

VAR
  cp: SIGNAL; (* current process *)
  main: SIGNAL; (* main program descriptor *)
  clk, intbyclock: PROCESS;
  StorageForMain: ProcessDescriptor;
  WSP: ARRAY [0..100] OF WORD; (* work space for clock process *)
  PCount: CARDINAL; (* number of active processes ==
    main_process + starts - halts *)
  Pnum: CARDINAL; (* count of all started processes *)
  clktrap[ClockTrapAddress], clkvect[ClockVectorAddress],
    savtrap, savvect: WORD;
    (* for saving and restoring the system clock trap vector *)
  running: BOOLEAN; (* indicates whether clock is running *)
  schedStatus: INTEGER;
    (* see notes for procedure StatusOfScheduler *)

PROCEDURE StatusOfScheduler() : INTEGER;
  (* indicates current state of the scheduling process
    InitState == 0 == never started
    RunState == -1 == running normally
    ProcHaltState == 1 == halted by HaltProcess
    SchedHaltState == 2 == halted by outside call to HaltScheduler
    InfiniteWaitState == 3 == halted by error call from Clock
    *)
  BEGIN
    RETURN schedStatus
  END StatusOfScheduler;

PROCEDURE ProcessID() : CARDINAL;
  BEGIN
RETURN cp^.ID
END ProcessID;

PROCEDURE start(P: PROC; A: ADDRESS; n: CARDINAL);
(* start P with workspace A of length n *)
VAR
t: SIGNAL;
workaddress: ADDRESS;
worksize: CARDINAL;
BEGIN
(* if not running, get room for the main to be stored,
and prepare to start the clock process *)
IF NOT running THEN
INC(PCount);
InitScheduler
END;
INC(PCount);
INC(Pnum);
t := A;
WITH t^ DO
next := cp^.next; (* put new process in the ring *)
cp^.next := t;
proc := P; (* requested PROC to run under 'job' *)
ready := TRUE; (* mark it as ready to run *)
ID := Pnum;
queue := NIL;
workaddress := A + TSIZE(ProcessDescriptor);
worksize := n - TSIZE(ProcessDescriptor);
NEWPROCESS(job,workaddress,worksize,pp)
END;
(* if not running, start up the clock processes *)
IF NOT running THEN TRANSFER(intbyclock,clk) END
END start;

PROCEDURE job; (* this is the actual running process *)
VAR PSW[PSWAddress]: INTEGER;
BEGIN
PSW := 0; (* set interrupt level to zero *)
cp^.proc; (* call the desired PROC *)
HaltProcess
END job;

PROCEDURE signal( VAR s: SIGNAL );
(* mark first process waiting for s as ready -- if there
are no waiting process, do nothing *)
BEGIN
IF s # NIL THEN
s^.ready := TRUE;
s := s^.queue
END
END signal;
PROCEDURE wait(VAR s: SIGNAL);
    (* wait until a signal on s is received — if the
    Clock is not running, the process is put in the queue,
    but nothing is really done. The wait queue is FIFO. *)
    VAR this, next: SIGNAL;
BEGIN (* insert current process at end of queue s *)
    IF s = NIL THEN
        s := cp
    ELSE
        this := s;
        LOOP (* find end of queue *)
            next := this''.queue;
            IF next = NIL THEN EXIT END;
            this := next
        END;
        this~.queue := cp
    END;
    cp^.ready := FALSE; (* mark process as not ready to run *)
    cp^.queue := NIL; (* flag as end of waiting queue *)
    (* if running, get the next process in the queue by
    TRANSFERing to Clock, otherwise ignore the wait...
    this prevents errors caused by waits when
    there is nothing to signal and halt the wait *)
    IF running THEN TRANSFER(intbyclock,clk) END
END wait;

PROCEDURE waited(VAR s: SIGNAL):BOOLEAN;
    (* return the status of the wait queue for s *)
BEGIN
    IF s=NIL THEN RETURN FALSE ELSE RETURN TRUE END
END waited;

PROCEDURE initsignal(VAR s: SIGNAL);
    (* Initialization of a SIGNAL *)
BEGIN
    s := NIL
END initsignal;

PROCEDURE Clock;
    (* this procedure acts as a clock, ticking 60 times per sec
    — it causes processes in the queue loop defined by `next'
    to be interleaved based upon the clock interrupt. Certain
    possible errors are detected. Some of this error detection
    is unusual for a scheduler clock interrupt, but are necessary
    as a part of an instructional package where concurrency
    simulation runs under another operating system. *)
    VAR
        temp: SIGNAL;
BEGIN
/* get the clock process started — the clock will only be called by start, and only when it is not already running *)
running := TRUE;
schedStatus := RunState;
LOOP (* once started, this loops until the clock interrupt vector is reset by HaltScheduler *)
IOTRANSFER(clk,intbyclock,ClockTrapAddress);
temp := cp;
REPEAT
    temp := temp^.next
    UNTIL temp^.ready OR (temp=cp);
(* if there are no ready processes, then an error has occurred in the user program — all processes are waiting and there will never be a signal — the scheduler clock will be halted and control returned to the main process. *)
IF (temp=cp) & (NOT cp^.ready) THEN
    schedStatus := InfiniteWaitState;
    HaltScheduler
END;
cp^.pp := intbyclock; (* save the current process *)
cp := temp; (* get the next process pointer *)
intbyclock := cp^.pp (* prepare to start the next process *)
END
END Clock;

PROCEDURE HaltProcess;
BEGIN
    cp^.ready := FALSE; (* current process is no longer runnable *)
    IF PCount>2 THEN (* there are more process to be executed *)
        DEC(PCount)
    ELSE (* if there is only one process running (plus main) *)
        (* in this case, the scheduler is no longer required, and should be halted in order to prevent possible errors *)
        schedStatus := ProcHaltState;
        HaltScheduler
    END;
    (* if running, get the next process in the queue by TRANSFERing to Clock, otherwise ignore the halt... this prevents errors caused by halts when the clock is not running. *)
    IF running THEN TRANSFER(intbyclock,clk) END
END HaltProcess;

PROCEDURE InitScheduler;
(* sets up the main process with ID=0 *)
BEGIN
    cp := ADDRESS(ADR(StorageForMain)); (* NEW(cp) *)
    main := cp;
WITH main^ DO
   ID := 0;  (* ID for main is always 0 *)
   ready := TRUE;
   next := cp;  (* make the run queue into a loop *)
   queue := NIL
END;
NEWPROCESS(Clock,ADR(WSP),SIZE(WSP),clk)
END InitScheduler;

PROCEDURE HaltScheduler;
VAR
   haltS: PROCESS;
BEGIN
   (* if running, shut down the clock by restoring the
     original interrupt vector, otherwise do nothing. *)
   IF running THEN
      clktrap := savtrap; ctrlvect := savvect;
      running := FALSE;
      IF schedStatus<RunState THEN
         schedStatus := SchedHaltState
      END;
      PCount := 0;
      IF cp^.ID#0 THEN (* if not currently in the main *)
         TRANSFER(haltS,main^..pp)
      END
   END
END HaltScheduler;

PROCEDURE P(VAR s: SEMAPHORE);
   (* the P operation on a SEMAPHORE is similar to a wait
      operation on a SIGNAL. All processes waiting for the
      same SEMAPHORE are maintained in a FIFO queue. *)
VAR this, next: SIGNAL;
BEGIN
   IF s^.count>0 THEN (* no wait if the count > 0 *)
      DEC(s^.count) (* and continue *)
   ELSE (* put the process into the queue *)
      IF s^.semqueue = NIL THEN
         s^.semqueue := cp
      ELSE
         this := s^.semqueue;
         LOOP (* find end of queue *)
            next := this^.queue;
            IF next = NIL THEN EXIT END ;
            this := next
         END ;
         this^.queue := cp
      END ;
      cp^.ready := FALSE;
      cp^.queue := NIL;
   END
END P;
(* if running, get the next process in the queue by
TRANSFERing to Clock, otherwise ignore the wait...
this prevents errors caused by waits when
there is nothing to signal and halt the wait *)
IF running THEN TRANSFER(intbyclock,clk) END
END

PROCEDURE V( VAR s: SEMAPHORE );
(* the V operation is similar to the signal operation
for a SIGNAL *)
BEGIN
  IF s^.semqueue = NIL THEN (* no process to start *)
    INC(s^.count)
  ELSE (* start the next waiting process *)
    s^.semqueue^.ready := TRUE;
    s^.semqueue := s^.semqueue^.queue
  END
END V;

PROCEDURE initsemaphore(VAR s: SEMAPHORE; initvalue: CARDINAL);
(* Initialization of a SEMAPHORE *)
BEGIN
  ALLOCATE(s, TSIZE(SemaphoreDescriptor));
  s^.count := initvalue;
  s^.semqueue := NIL
END initsemaphore;

PROCEDURE TraceOutput(Tfile: ChanNum; filename: FileName);
(* This is a dummy version. *)
BEGIN
END TraceOutput;

BEGIN
  savtrap := elktrap; savvect := clkvect;
  running := FALSE;
  schedStatus := InitState;
  PCount := 0;
  Pnum := 0
END PreemptiveScheduler.
IMPLEMENTATION MODULE PreemptiveScheduler [6];
("$T-$, $S-$")

(* original version by N. Wirth
 modified by Dr. John Barr
 modified and partially re-written by Phil Rosine
 Jan 84

This module implements a pre-emptive scheduler. The Clock
routine will interrupt all processes and start the next
process in the ready queue if possible.
*)

(* modified 12 Apr 84 — per

Added a trace feature consisting of the procedure
TraceOutput.

There are two IMPLEMENTATION MODULES: one with the trace
feature implemented and one with a dummy trace. This
Version has the trace implemented.
*)

(******************************************************************************
*# #*##-###############*###*# #*
*# The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and *# all parts of the Monitor Pre-processor program. *#
*# Copyright 1984. Philip E. Rosine *#
******************************************************************************)

FROM SYSTEM IMPORT WORD, PROCESS, ADDRESS, ADR, SIZE, TSIZE,
NEWPROCESS, TRANSFER, IOTRANSACTION, LISTEN, SYSRESET;
FROM Storage IMPORT ALLOCATE;
FROM FInOut IMPORT FOpenOutput,FCloseOutput,
FWriteCard,FWriteOct,FWrite,FWriteLn,ChanNum,FDone;
FROM Files IMPORT FileName;

CONST
ClockTrapAddress = 100B;
ClockVectorAddress = ClockTrapAddress + 2;
PSWAddress = 177776B;
TraceSize = 300;

(* the following define the return codes for the
 Scheduler Status — see notes with StatusOfScheduler *)
RunState = -1; InitState = 0; ProcHaltState = 1;
SchedHaltState = 2; InfiniteWaitState = 3;
(* the following define the trace description codes — see the notes with TraceOutput *)

startup = 'I'; change = 'C'; halt = 'H'; Pend = 'X';
Sigwait = 'W'; Sigsignal = 'S'; SemP = 'P'; SemV = 'V';

TYPE

SIGNAL = POINTER TO ProcessDescriptor;
ProcessDescriptor =
    RECORD
        pp: PROCESS;
        proc: PROC; (* procedure to run *)
        ID: CARDINAL; (* PCount when process started *)
        ready: BOOLEAN; (* indicates the run status of the process *)
        next: SIGNAL; (* ties the descriptors in the ring *)
        queue: SIGNAL (* ties processes waiting for the same signal *)
    END;

SEMAPHORE = POINTER TO SemaphoreDescriptor;
SemaphoreDescriptor = RECORD
    count: CARDINAL;
    semqueue: SIGNAL
END;

TraceRecord =
    RECORD
        CurrentID: CARDINAL; (* ID of current process *)
        PC: CARDINAL; (* program counter for current process *)
        Action: CHAR; (* reason scheduler was invoked *)
        Variable: CARDINAL; (* address of procedure, SIGNAL, or
          SEMAPHORE involved in action *)
        NextID: CARDINAL (* ID of process starting to execute *)
    END;

STACK = (* locations of registers in the PROCESS stack *)
    RECORD
        R1, R0, R5, R3, R4, R2, PC, PS: WORD
    END;

VAR

cp: SIGNAL; (* current process *)
main: SIGNAL; (* main program descriptor *)
clk, intbyclock: PROCESS;
StorageForMain: ProcessDescriptor;
WSP: ARRAY [0..100] OF WORD; (* work space for clock process *)
PCount: CARDINAL; (* number of active processes ==
    main_process + starts - halts *)
Pnum: CARDINAL; (* count of all started processes *)
clktrap[ClockTrapAddress], clkvect[ClockVectorAddress],
savtrap, savvect: WORD;
(* for saving and restoring the system clock trap vector *)
running: BOOLEAN; (* indicates whether clock is running *)
(* running is also used to prevent interruption of the
  trace feature while it is writing to the file *)
schedStatus: INTEGER;

(* see notes for procedure StatusOfScheduler *)
TRACE: ARRAY [0..TraceSize-1] OF TraceRecord;
BlankTrace: TraceRecord;
Tcount: CARDINAL; (* counter into TRACE *)
traced: BOOLEAN;

(* indicator that a process change has been traced *)
stack: POINTER TO STACK;

(* for locating the PC of an interrupted process *)

PROCEDURE StatusOfScheduler() : INTEGER;
(* indicates current state of the scheduling process
InitState = 0 == never started
RunState = -1 == running normally
ProcHaltState = 1 == halted by HaltProcess
SchedHaltState = 2 == halted by outside call to HaltScheduler
InfiniteWaitState = 3 == halted by error call from Clock *)
BEGIN
  RETURN schedStatus
END StatusOfScheduler;

PROCEDURE ProcessID() : CARDINAL;
BEGIN
  RETURN cp^.ID
END ProcessID;

PROCEDURE start(P: PROC; A: ADDRESS; n: CARDINAL);
(* start P with workspace A of length n *)
VAR
  t: SIGNAL;
  workaddress: ADDRESS;
  worksize: CARDINAL;
BEGIN
  (* if not running, get room for the main to be stored,
  and prepare to start the clock process *)
  IF NOT running THEN
    INC(PCount);
    InitScheduler
  END;
  INC(PCount);
  INC(Pnum);
  t := A;
  WITH t^ DO
    next := cp^.next; (* put new process in the ring *)
    cp^.next := t;
    proc := P; (* requested PROC to run under 'job' *)
    ready := TRUE; (* mark it as ready to run *)
ID := Pnum;
queue := NIL;
workaddress := A + TSIZE(ProcessDescriptor);
worksize := n - TSIZE(ProcessDescriptor);
NEWPROCESS(job,workaddress,worksize,pp);
WITH TRACE[Tcount] DO (* enter the trace *)
   CurrentID := ID;
   PC := 0;
   Action := startup;
   Variable := CARDINAL(P);
   NextID := cp^.ID
END;
Tcount := (Tcount+1) MOD TraceSize
END;
(* if not running, start up the clock processes *)
IF NOT running THEN TRANSFER(intbyclock,clk) END
END start;

PROCEDURE job; (* this is the actual running process *)
VAR PSW[PSWAddress]: INTEGER;
BEGIN
   PSW := 0; (* set interrupt level to zero *)
   cp^.proc; (* call the desired PROC *)
   HaltProcess
END job;

PROCEDURE signal( VAR s: SIGNAL );
(* mark first process waiting for s as ready -- if there
   are no waiting process, do nothing *)
BEGIN
   IF s # NIL THEN
      s^.ready := TRUE;
      s := s^.queue
   END;
   WITH TRACE[Tcount] DO (* enter the trace *)
      CurrentID := cp^.ID;
      PC := 0;
      Action := Sigsignal;
      Variable := CARDINAL(ADR(s));
      NextID := CurrentID
   END;
   Tcount := (Tcount+1) MOD TraceSize
END signal;

PROCEDURE wait(VAR s: SIGNAL);
(* wait until a signal on s is received -- if the
   Clock is not running, the process is put in the queue,
   but nothing is really done. The wait queue is FIFO. *)
VAR this, next: SIGNAL;
BEGIN (* insert current process at end of queue s *)
   IF s = NIL THEN
s := cp
ELSE
   this := s;
END
LOOP
   next := this''.queue;
   IF next = NIL THEN EXIT END;
   this := next
END;
this''.queue := cp
END;
cp^.ready := FALSE; (* mark process as not ready to run *)
cp^.queue := NIL; (* flag as end of waiting queue *)
WITH TRACE[Tcount] DO (* enter the trace *)
   CurrentID := cp^.ID;
   PC := 0;
   Action := Sigwait;
   Variable := CARDINAL(ADR(s))
END;
   (* if running, get the next process in the queue by
    transferring to Clock, otherwise ignore the wait...
    this prevents errors caused by waits when
    there is nothing to signal and halt the wait *)
IF running THEN
   traced := TRUE;
   TRANSFER(intbyclock,clk)
ELSE
   TRACE[Tcount].NextID := cp^.ID;
   Tcount := (Tcount+1) MOD TraceSize
END
END wait;

PROCEDURE waited(VAR s: SIGNAL ):BOOLEAN;
   (* return the status of the wait queue for s *)
BEGIN
   IF s=NIL THEN RETURN FALSE ELSE RETURN TRUE END
END waited;

PROCEDURE initsignal(VAR s: SIGNAL);
   (* Initialization of a SIGNAL *)
BEGIN
   s := NIL
END initsignal;

PROCEDURE Clock;
   (* this procedure acts as a clock, ticking 60 times per sec
   -- it causes processes in the queue loop defined by "next"
   to be interleaved based upon the clock interrupt. Certain
   possible errors are detected. Some of this error detection
   is unusual for a scheduler clock interrupt, but are necessary
   as a part of an instructional package where concurrency

simulation runs under another operating system. *)
VAR
temp: SIGNAL;
BEGIN
(* get the clock process started — the clock will only be
called by start, and only when it is not already running *)
running := TRUE;
schedStatus := RunState;
LOOP (* once started, this loops until the clock
interrupt vector is reset by HaltScheduler *)
  IOTRANSFER(clk,intbyclock,ClockTrapAddress);
  IF running THEN (* get the next process — otherwise the
  trace feature is writing to the file and must
  not be interrupted *)
    temp := cp;
    REPEAT
      temp := temp^.next
    UNTIL temp^.ready OR (temp=cp);
    (* if there are no ready processes, then an error has
    occurred in the user program — all processes are
    waiting and there will never be a signal — the
    scheduler clock will be halted and control
    returned to the main process. *)
  IF (temp=cp) & (NOT cp^.ready) THEN
    schedStatus := InfiniteWaitState;
    HaltScheduler
  END;
  cp^.pp := intbyclock; (* save the current process *)
  WITH TRACE[Tcount] DO (* enter the trace *)
    NextID := temp^.ID;
    IF traced THEN traced := FALSE
    ELSE
      CurrentID := cp^.ID;
      stack := ADDRESS(cp^.pp);
      PC := CARDINAL(stack^.PC);
      Action := change;
      Variable := 0
    END
  END
  Tcount := (Tcount+1) MOD TraceSize;
  cp := temp; (* get the next process pointer *)
  intbyclock := cp^.pp (* prepare to start the next process *)
END (* of IF running check *)
END (* of IOTRANSFER LOOP *)
END Clock;

PROCEDURE HaltProcess;
BEGIN
  cp^.ready := FALSE; (* current process is no longer runnable *)
  WITH TRACE[Tcount] DO (* enter the trace *)
CurrentID := cp^.ID;
PC := 0;
Action := Pend;
Variable := 0
END;
IF PCount>2 THEN (* there are more process to be executed *)
DEC(PCount)
ELSE (* if there is only one process running (plus main) *)
(* in this case, the scheduler is no longer required, and should be halted in order to prevent possible errors *)
schedStatus := ProcHaltState;
TRACE[Tcount].NextID := 0;
Tcount := (Tcount+1) MOD TraceSize;
HaltScheduler
END;
(* if running, get the next process in the queue by TRANSFERing to Clock, otherwise ignore the halt... this prevents errors caused by halts when the clock is not running. *)
IF running THEN
  traced := TRUE;
  TRANSFER(int by c1ock,clk)
ELSE
  TRACE[Tcount].NextID := cp^.ID;
  Tcount := (Tcount+1) MOD TraceSize
END
END HaltProcess;

PROCEDURE InitScheduler;
(* sets up the main process with ID=0 *)
BEGIN
  cp := ADDRESS(ADR(StorageForMain)); (* NEW(cp) *)
  main := cp;
  WITH main^ DO
    ID := 0; (* ID for main is always 0 *)
    ready := TRUE;
    next := cp;  (* make the run queue into a loop *)
    queue := NIL
  END;
  NEWPROCESS(Clock,ADR(WSP),SIZE(WSP),clk)
END InitScheduler;

PROCEDURE HaltScheduler;
VAR
  haltS: PROCESS;
BEGIN
  (* if running, shut down the clock by restoring the original interrupt vector, otherwise do nothing. *)
  WITH TRACE[Tcount] DO (* enter the trace *)
CurrentID := cp^.ID;
PC := 0;
Action := halt;
Variable := 0;
NextID := 0
END;
Tcount := (Tcount+1) MOD TraceSize;
IF running THEN
  clktrap := savtrap; clkvect := savvect;
  running := FALSE;
  IF schedStatus<RunState THEN
    schedStatus := SchedHaltState
  END;
PCount := 0;
IF cp^.ID#0 THEN (* if not currently in the main *)
  TRANSFER(haltS,main^.pp)
END
END
END HaltScheduler;

PROCEDURE P(VAR s: SEMAPHORE);
(* the P operation on a SEMAPHORE is similar to a wait
operation on a SIGNAL. All processes waiting for the
same SEMAPHORE are maintained in a FIFO queue. *)
VAR this, next: SIGNAL;
BEGIN
  IF s^.count>0 THEN (* no wait if the count > 0 *)
    DEC(s^.count); (* and continue *)
    WITH TRACE[Tcount] DO (* enter the trace *)
      CurrentID := cp^.ID;
      PC := 0;
      Action := SemP;
      Variable := CARDINAL(ADR(s));
      NextID := cp^.ID
    END;
    Tcount := (Tcount+1) MOD TraceSize
  ELSE (* put the process into the queue *)
    IF s^.semqueue = NIL THEN
      s^.semqueue := cp
    ELSE
      this := s^.semqueue;
      LOOP (* find end of queue *)
        next := this^.queue;
        IF next = NIL THEN EXIT END ;
      this := next
    END;
    this^.queue := cp
  END;
  cp^.ready := FALSE;
  cp^.queue := NIL;
(* if running, get the next process in the queue by
TRANSFERing to Clock, otherwise ignore the wait...
this prevents errors caused by waits when
there is nothing to signal and halt the wait *)
WITH TRACE[Tcount] DO (* enter the trace *)
  CurrentID := cp^.ID;
  PC := 0;
  Action := SemP;
  Variable := CARDINAL(ADR(s))
END;
IF running THEN
  traced := TRUE;
  TRANSFER(intbyclock,clk)
ELSE
  TRACE[Tcount].NextID := cp^.ID;
  Tcount := (Tcount+1) MOD TraceSize
END
END P;

PROCEDURE V(VAR s: SEMAPHORE);
(* the V operation is similar to the signal operation
for a SIGNAL *)
BEGIN
  IF s^.semqueue = NIL THEN (* no process to start *)
    INC(s^.count)
  ELSE (* start the next waiting process *)
    s^.semqueue^.ready := TRUE;
    s^.semqueue := s^.semqueue^.queue
  END;
  WITH TRACE[Tcount] DO (* enter the trace *)
    CurrentID := cp^.ID;
    PC := 0;
    Action := SemV;
    Variable := CARDINAL(ADR(s));
    NextID := CurrentID
  END;
  Tcount := (Tcount+1) MOD TraceSize;
END V;

PROCEDURE initsemaphore(VAR s: SEMAPHORE; initvalue: CARDINAL);
(* Initialization of a SEMAPHORE *)
BEGIN
  ALLOCATE(s, TSIZE(SemaphoreDescriptor));
  s^.count := initvalue;
  s^.semqueue := NIL
END initsemaphore;

PROCEDURE TraceOutput(Tfile: ChanNum; filename: FileName);
(* This procedure causes the history of the Preemptive-Scheduler (for approximately the last 5 seconds) to be written out to file `filename`. The trace history has the form:

ProcessID/PC/Action/Variable/NextID

where ProcessID is the CARDINAL number assigned to the current process by the scheduler. PC is the address of the next instruction (i.e., the Program Counter). PC only has validity if the Action code is C, otherwise it will be zero. Action is a code as follows:

I = initial entry of process into the process chain
C = change of process due to time-slicing
H = halted by a call to HaltScheduler
X = halted due to normal procedure end
W = caused to wait on a SIGNAL
S = released by a signal on a SIGNAL
P = caused to wait on a SEMAPHORE
V = released by a V on a SEMAPHORE

Variable is the address of the procedure being started as a process if Action is `I`, and is the address of the SIGNAL or SEMAPHORE if Action is `W`, `S`, `P`, or `V`; in all other cases it will be 0 and has no meaning. NextID is the number of the next process to be executed by the scheduler (all of these shifts are recorded when the scheduler is trying to switch processes — ProcessID is the one being switched from and NextID is the one being switched to: they may be the same). NextID will not have any meaning if Action is `S` or `V`, and will be set to CurrentID in these cases. *)

(* NOTE:
There are two implementation modules for this definition: the file PREEMP.* has a dummy trace feature and procedure TraceOutput has no action. The file TRACES.* has the actual trace feature implemented. To use the package, compile normally, and link with the /q switch if the trace feature is required. When prompted for PreemptiveScheduler, enter `traces` in order to link in the trace package.
*)

CONST slash = `/`;
VAR
  i: CARDINAL; (*local counter*)
  temptrap, tempvect: WORD; (*to save the current clock vector*)
BEGIN
  temptrap := clktrap; tempvect := clkvect;
  clktrap := savtrap; clkvect := savvect;
  (* ensure that the scheduler doesn't interrupt *)
  FOpenOutput(Tfile, filename);
  IF FDone THEN
    i := Tcount;
  REPEAT
IF TRACE[i].Action#BlankTrace.Action THEN
  WITH TRACE[i] DO
    FWriteCard(Tfile,CurrentID,5);
    FWrite(Tfile,slash);
    FWriteOct(Tfile,PC,6);
    FWrite(Tfile,"B");
    FWrite(Tfile,slash);
    FWrite(Tfile,Action);
    FWrite(Tfile,slash);
    FWriteOct(Tfile,Variable,6);
    FWrite(Tfile,"B");
    FWrite(Tfile,slash);
    FWriteCard(Tfile,NextID,5)
  END; (* WITH *)
  FWriteLn(Tfile);
END; (* IF not BlankTrace *)
i := (i+1) MOD TraceSize
UNTIL i=Tcount;
FCloseOutput(Tfile);
FOR Tcount := 0 TO TraceSize-1 DO
  TRACE[Tcount] := BlankTrace
END;
Tcount := 0
END; (* IF FDone *)
clktrap := temptrap; clkvect := tempvect
(* restore the clock interrupt *)
END TraceOutput;

BEGIN
  savtrap := clktrap; savvect := clkvect;
  running := FALSE;
  schedStatus := InitState;
PCount := 0;
Pnum := 0;
traced := FALSE;
WITH BlankTrace DO
  CurrentID := 0;
  PC := 0;
  Action := ' ';
  Variable := 0;
  NextID := 0
END;
FOR Tcount := 0 TO TraceSize-1 DO
  TRACE[Tcount] := BlankTrace
END;
Tcount := 0
END PreemptiveScheduler.
DEFINITION MODULE StrLibrary;

(* written by Phil Rosine
26 Mar 84

This module is a library of several functions which assist in handling strings. A string (in general) is an array of characters. More specifically, a string is an array of characters with a NULL (OC) marking the end of the string. The string array should be defined as:

TYPE String = ARRAY [0..StringSize] OF CHAR;
where StringSize is a CARDINAL constant. These functions will handle any StringSize.

Most of these functions are modeled and named after similar functions supplied by the UNIX C library.
*)

EXPORT QUALIFIED strcmp, strncmp, strncpy;

PROCEDURE strcmp(stringA,stringB : ARRAY OF CHAR) : BOOLEAN;
(* This is like the UNIX strcmp function which compares two strings of characters to see if they are equal; this function simply returns TRUE if they are and FALSE if not. A string is a null terminated ARRAY OF CHAR. This algorithm may not return meaningful values for work for strings which are not null terminated. In particular, it will not work for quoted strings which are within the source code. To use 'quoted' strings, they must be copied into a variable using strncpy (which will work with a quoted string) and the last character set to OC. See strncmp. *)

PROCEDURE strncmp(stringA,stringB : ARRAY OF CHAR; startA, startB,length : CARDINAL) : BOOLEAN;
(* This is version of strcmp which compares two character strings starting at the appropriate locations and comparing length characters. The strings need not be null terminated. *)

PROCEDURE strncpy(stringA : ARRAY OF CHAR; VAR stringB : ARRAY OF CHAR; startA,startB,length : CARDINAL);
(* This is like the UNIX strncpy function which copies length characters from stringA starting at character startA into stringB starting at character startB. This function does not
ensure that either ARRAY is truly a 'string', ie, that it terminates with a null; if stringB needs to be null terminated, the calling program must ensure that it is. *)

END StrLibrary.
IMPLEMENTATION MODULE StrLibrary;

(* written by Phil Rosine
22 Mar 84

This module is a library of several functions which assist in handling strings. A string (in general) is an array of characters. More specifically, a string is an array of characters with a NULL (0C) marking the end of the string. The string array should be defined as:

    TYPE String = ARRAY [0..StringSize] OF CHAR;

where StringSize is a CARDINAL constant. These functions will handle any StringSize.

Most of these functions are modeled and named after similar functions supplied by the UNIX C library.
*)

PROCEDURE strcmp(stringA,stringB : ARRAY OF CHAR) : BOOLEAN;
(* This is like the UNIX strcmp function which compares two strings of characters to see if they are equal; this function simply returns TRUE if they are and FALSE if not. A string is a null terminated ARRAY OF CHAR. This algorithm may not return meaningful values for work for strings which are not null terminated. In particular, it will not work for quoted strings which are within the source code. To use quoted strings, they must be copied into a variable using strncpy (which will work with a quoted string) and the last character set to 0C. See strncmp. *)

VAR
    testvalue : BOOLEAN;
    i : CARDINAL; (* local counter *)
BEGIN
    i := 0;
    testvalue := (stringA[i]=stringB[i]);
    WHILE (stringA[i]#0C) & (stringB[i]#0C) & testvalue DO
        INC(i);
        testvalue := ((i<=HIGH(stringA)) & (i<=HIGH(stringB))
            & (stringA[i]=stringB[i]));
    END;
RETURN testvalue
END strcmp;
PROCEDURE strncmp(stringA,stringB : ARRAY OF CHAR;
    startA,startB,length : CARDINAL) : BOOLEAN;
    (* This is version of strcmp which compares two character
        strings starting at the appropriate locations and comparing
        length characters. The strings need not be null terminated.
        *)
VAR
    testvalue : BOOLEAN;
    i, j, stop : CARDINAL;
BEGIN
    i := startA;
    j := startB;
    stop := startA + length - 1;
    testvalue := ((stop<=HIGH(stringA)) &
        ((startB+length-1)<=HIGH(stringB)) & (stringA[i]=stringB[j]));
    WHILE testvalue & (i<stop) DO
        INC(i); INC(j);
        testvalue := (stringA[i]=stringB[j])
    END;
    RETURN testvalue
END strncmp;

PROCEDURE strncpy(stringA : ARRAY OF CHAR;
    VAR stringB : ARRAY OF CHAR;
    startA,startB,length : CARDINAL);
    (* This is like the UNIX strncpy function which copies length
        characters from stringA starting at character startA into
        stringB starting at character startB. This function does not
        ensure that either ARRAY is truly a "string", ie, that it
        terminates with a null; if stringB needs to be null
        terminated, the calling program must ensure that it is. *)
    (* tested 22 Mar 84 -- per -- works OK *)
VAR
    i, A, B : CARDINAL; (* local counters *)
BEGIN
    A := startA; B := startB; i := 1;
    WHILE (i<=length) & (A<=HIGH(stringA)) & (B<=HIGH(stringB)) DO
        INC(A); INC(B); INC(i);
    END
END strncpy;

BEGIN
    (* no initialization required *)
END StrLibrary.
MODUlE MONITR;

(* written by Phil Rosine
20 Mar 84

This is the monitor pre-processor main program. The
monitor pre-processor prompts a user for a file name,
and processes that file to produce an IMPLEMENTATION
MODULE which acts as a monitor. The required input
files are the monitor file (XXXXXX.MON), the definition
file (XXXXXX.DEF), and the output file is a module
(XXXXXX.MOD); all of these have the filename as given
by the user (XXXXXX). Additionally a data file
(MONITR.DAT) is required. See the MONITR User's Guide
for information on what the input files contain and how
to use this program.*
)

(**************************************************************************
** # # * # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # **
** Copyright 1984. Philip E. Rosine **
** # # **
** # # The copyright covers both DEFINITION and IMPLEMENTATION **
** MODULES of PreemptiveScheduler, and **
** all parts of the Monitor Pre-processor program. **
** # # **
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # **
**************************************************************************)

FROM MData IMPORT NameList;
FROM MFileHandler IMPORT OpenFiles, CloseFiles;
FROM MGetDefs IMPORT GetDefsMain;
FROM MConvert IMPORT ConvertMain;

VAR
    Err : BOOLEAN;
    (* return code from file opening *)
    ExpCondList : NameList;
    (* list of exported procedures declared by XXXXXX.DEF *)

BEGIN
    OpenFiles(Err);
    IF NOT Err THEN
        GetDefsMain(ExpCondList);
        ConvertMain(ExpCondList)
    END;
    CloseFiles
END MONITR.
DEFINITION MODULE MData;

(* written by Phil Rosine  
  20 Mar 84 

  This is a data module for the monitor pre-processor. It 
  exports and initializes some of the major data items 
  needed by the pre-processor. *)

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Copyright 1984. Philip E. Rosine #*
# # #
# The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and #
# all parts of the Monitor Pre-processor program. *
# #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
**************************************************************************************************

EXPORT QUALIFIED Name, ParseToken, Word, DATfile, 
    DEFfile, MODfile, MONfile, FileNum, NameList;

CONST
    DATfile = 1;
    DEFfile = 2;
    MODfile = 3;
    MONfile = 4;
    NameSize = 40;
    StringSize = 132;

TYPE
    NameList = POINTER TO Name;
    Name = RECORD
        NameString : ARRAY [0..NameSize] OF CHAR;
        Length : CARDINAL;
        Next : NameList
    END;
    ParseToken = (const,type,var,procedure,comment,begin,end, 
                  export,return,other);
    Word = RECORD
        CharString : ARRAY [0..StringSize] OF CHAR;
        FirstChar : CARDINAL;
        Length : CARDINAL
    END;
    FileNum = [1..4];

END MData.
IMPLEMENTATION MODULE MData;

(* written by Phil Rosine
20 Mar 84

This is a data module for the monitor pre-processor. It exports and initializes some of the major data items needed by the pre-processor. *)

(* do nothing *)
END MData.
DEFINITION MODULE MFileHandler;

(* written by Phil Rosine
20 Mar 84

This handles file opening and closing for the monitor pre-processor. It also does part of the file I/O for the MONITR program. *)

(*********************************************************************
* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
* # Copyright 1984. Philip E. Rosine #*
* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
* # The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and #*
* # all parts of the Monitor Pre-processor program. #*
* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
*********************************************************************)

FROM MData IMPORT Word, FileNum;

EXPORT QUALIFIED OpenFiles, CloseFiles, GetWord;

PROCEDURE OpenFiles(VAR ErrorCode : BOOLEAN);

(* Opens all the files required for the pre-processor and returns ErrorCode FALSE if no errors; otherwise prints error messages and returns ErrorCode TRUE. OpenFiles prompts the user for the filename to be used for opening DEFFile, MONfile, and MODfile. *)

PROCEDURE CloseFiles;

(* Closes the all files. *)

PROCEDURE GetWord(File : FileNum; VAR word : Word);

(* Gets the next word from the designated file (either MONfile or DEFFile). *)

END MFileHandler.
IMPLEMENTATION MODULE MFileHandler;

(* written by Phil Rosine
21 Mar 84

This handles file opening and closing for the monitor pre-processor. It also does part of the file I/O for the MONITR program. *)

(*---------------------------------------------------------
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Copyright 1984. Philip E. Rosine #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and
# all parts of the Monitor Pre-processor program. #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# *******************************************************
#--------------------------------------------------------)

FROM MData IMPORT DATfile, DEFFile, MONfile, MODfile,
FileNum, Word;
FROM FInOut IMPORT FOpenInput, FOpenOutput, FCloselnput,
FCloseOutput, Read, FDone;
FROM InOut IMPORT WriteString, WriteLn;
FROM FileNames IMPORT ReadFileName;
FROM Files IMPORT FileName;

CONST
    tab = #11C;
    If = #12C;
    cr = #25C;
    space = #40C;

PROCEDURE OpenFiles(VAR ErrorCode : BOOLEAN);
(* Opens all the files required for the pre-processor and returns ErrorCode FALSE if no errors; otherwise prints error messages and returns ErrorCode TRUE. OpenFiles prompts the user for the filename to be used for opening DEFFile, MONfile, and MODfile. *)

(* TESTED: 21 Mar 84 -- per -- works OK.
NOTE: if the output file (XXXXXX.MON) is protected it will still be renamed XXXXXX.QOD and the first execution of MONITR will work. The second execution will result in not being able to open the output file and ErrorCode will be returned TRUE. *)
VAR
  Name : FileName;
BEGIN
  ErrorCode := FALSE;
  WriteString("Monitor filename (extension MON default) => ");
  ReadFileName(Name, 'DK XXXXXXMON');
  WriteLn;
    (* error in filename *)
    WriteLn;
    WriteString("INCORRECT INPUT FILE EXTENSION");
    WriteLn;
    ErrorCode := TRUE
  ELSE
    FOpenInput(MONfile, Name);
    IF NOT FDone THEN
      (* error opening MONfile *)
      WriteLn;
      WriteString("FILE NOT FOUND");
      WriteLn;
      ErrorCode := TRUE
    ELSE
      FOpenInput(DEFfile, Name);
      IF NOT FDone THEN
        (* error opening DEFfile *)
        WriteLn;
        WriteString("NO DEFINITION MODULE FOUND");
        WriteLn;
        ErrorCode := TRUE
      ELSE
        FOpenOutput(MODfile, Name);
        IF NOT FDone THEN
          (* error opening MODfile *)
          WriteLn;
          WriteString("COULD NOT OPEN OUTPUT FILE");
          WriteLn;
          ErrorCode := TRUE
        ELSE
          FOpenInput(DATfile, 'DK MONITRDAT');
          IF NOT FDone THEN
            FOpenInput(DATfile, 'SY MONITRDAT');
            IF NOT FDone THEN
              (* error opening DATfile *)
              WriteLn;
              WriteString("MONITR DATA FILE NOT FOUND");
              WriteLn;
              ErrorCode := TRUE
            END
          END
        END
      END
    END
  END
END
PROCEDURE CloseFiles;
(* Closes the all files. *)
(* TESTED: 21 Mar 84 — per — works OK. *)
BEGIN
FCloseInput(DATfile);
FCloseInput(DEFfile);
FCloseInput(MONfile);
FCloseOutput(MODfile)
END CloseFiles;

PROCEDURE GetWord(File :  FileNum; VAR word :  Word);
(* Gets the next word from the designated file (either MONfile or DEFfile). *)
(* TESTED: 21 Mar 84 — per — works OK. *)
VAR
count :  CARDINAL; (* local counter *)
BEGIN
count := 0;
WITH word DO

FRead(File,CharString[count]);
WHILE (CharString[count]=tab) OR (CharString[count]=lf) OR
(CharString[count]=cr) OR (CharString[count]=space) DO
INC(count);
FRead(File,CharString[count]);
END;
FirstChar := count;
WHILE ((CharString[count]>57C) & (CharString[count]<72C)) OR
((CharString[count]>100C) & (CharString[count]<133C)) OR
((CharString[count]>140C) & (CharString[count]<173C)) DO
(* this allows the digits 0..9, and the letters
A..Z,a..z to be processed *)
INC(count);
FRead(File,CharString[count]);
END;
Length := count - FirstChar;
CharString[count+1] := 0C
END
END GetWord;

BEGIN
(* no initialization needed *)
END MFileHandler.
DEFINITION MODULE MLibrary;

(* written by Phil Rosine
22 Mar 84

This module is a library of several functions which are
of more or less general use to the Monitor Pre-Processor.
*)

(********************************************************************
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #**
** Copyright 1984, Philip E. Rosine**
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #**
** The copyright covers both DEFINITION and IMPLEMENTATION**
** MODULES of PreemptiveScheduler, and**
** all parts of the Monitor Pre-processor program.**
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #**
********************************************************************)

FROM MData IMPORT Word, ParseToken, FileNum;

EXPORT QUALIFIED Token, CommentHandler, QuoteHandler,
WriteErrorHalt;

PROCEDURE Token(word : Word) : ParseToken;
(* This function checks to see if a word found in the input
file is a keyword which is in the ParseToken list and returns
the token for that word. If not, it returns the token
"other". *)

PROCEDURE CommentHandler(File : FileNum; write : BOOLEAN);
(* Reads comments (even if nested) from File and writes them
to the MODfile if write=TRUE. *)

PROCEDURE QuoteHandler(File:FileNum;write:BOOLEAN;Qtype:CHAR);
(* Reads quoted strings (even if nested) from File and writes
them to the MODfile if write=TRUE. Qtype is a character
indicating which type of quote (single or double) to check
for. *)

PROCEDURE WriteErrorHalt(stringl,string2,string3,string4 :
ARRAY OF CHAR);
(* This routine is used to handle terminal errors found by the
monitor pre-processor, usually involving end of file errors. *)
END MLibrary.
IMPLEMENTATION MODULE MLibrary;

(* written by Phil Rosine
22 Mar 84

This module is a library of several functions which are
of more or less general use to the Monitor Pre-Processor.
*)

(**************************************************************************
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
** Copyright 1984. Philip E. Rosine
**
** The copyright covers both DEFINITION and IMPLÉ-**
** MENTATION MODULES of PreemptiveScheduler, and
** all parts of the Monitor Pre-processor program.
**
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
**************************************************************************)

FROM MData IMPORT Word, ParseToken, FileNum, MODfile;
FROM StrLibrary IMPORT strcmp, strncmp, strncpy;
FROM FInOut IMPORT FWrite, FWriteString;
FROM InOut IMPORT Write, WriteString, WriteLn;
FROM MFileHandler IMPORT GetWord, CloseFiles;

CONST
    EOF = 0C; (* end of file marker *)
    BELL = 7C; (* beeps the terminal *)

TYPE
    String = ARRAY [0..9] OF CHAR;

VAR
    i : CARDINAL; (* local counter *)
    paren : BOOLEAN; (* used as a 'memory' cell for Token *)

PROCEDURE Token(word : Word) : ParseToken;

(* This function checks to see if a word found in the input
file is a keyword which is in the ParseToken list and returns
the token for that word. If not, it returns the token
'other'. *)

(* ParseToken is defined as follows:*)

ParseToken = (const,type,var,procedure,comment,begin,end,
            export,other);

(* In order to check for the start of a comment, this procedure
needs to know what the end character for the previous word

was. In particular, it needs to know if it was a left parenthesis; the BOOLEAN paren is used to provide a means of remembering this. *)

(* tested 22 Mar 84 — per — works OK *)

VAR
token : ParseToken;
BEGIN
  WITH word DO
    token := other;
    CASE Length OF
      9 : (* check for PROCEDURE *)
        IF strncmp(CharString,"PROCEDURE",FirstChar,0,9)
        THEN token := procedure END
      | 6 : (* check for EXPORT *)
        IF strncmp(CharString,"EXPORT",FirstChar,0,6)
        THEN token := export
          ELSIF strncmp(CharString,"RETURN",FirstChar,0,6)
        THEN token := return END
      | 5 : (* check for BEGIN or CONST *)
        IF strncmp(CharString,"BEGIN",FirstChar,0,5)
        THEN token := begin
          ELSIF strncmp(CharString,"CONST",FirstChar,0,5)
        THEN token := const END
      | 4 : (* check for TYPE *)
        IF strncmp(CharString,"TYPE",FirstChar,0,4)
        THEN token := type END
      | 3 : (* check for END or VAR *)
        IF strncmp(CharString,"END",FirstChar,0,3)
        THEN token := end
          ELSIF strncmp(CharString,"VAR",FirstChar,0,3)
        THEN token := var END
      | 0 : (* check for a comment *)
        IF paren THEN
          IF CharString[FirstChar]='*' THEN token := comment END
          END
        ELSE (* token := other *)
        END;
    paren := (CharString[FirstChar+Length]='(')
  END;
  RETURN token
END Token;

PROCEDURE CommentHandler(File : FileNum; write : BOOLEAN);
  (* Reads comments (even if nested) from File and writes them to the MODfile if write=TRUE. *)
  (* tested 23 Mar 84 — per — works OK *)

VAR
  word : Word;
BEGIN
IF write THEN FWrite(MODfile,'*') END;
GetWord(File,word);
LOOP
  IF word.CharString[word.FirstChar+word.Length]=EOF THEN
    (* error! *)
    WriteLn;
    WriteString("**** Unexpected end of file found ****");
    WriteLn;
    WriteString(" ** while reading a comment **");
    WriteLn;
    CloseFiles;
    HALT
  END;
  IF Token(word)=comment THEN
    CommentHandler(File,write); (* handle nested comments *)
    GetWord(File,word)
  ELSE
    IF write THEN FWriteString(MODfile,word.CharString) END;
    (* check to see if this is the end of the comment *)
    IF word.CharString[word.FirstChar+word.Length]="*" THEN
      GetWord(File,word);
      IF word.CharString[word.FirstChar+word.Length]=EOF THEN
        (* error! *)
        WriteLn;
        WriteString("**** Unexpected end of file found ****");
        WriteLn;
        WriteString(" ** while reading a comment **");
        WriteLn;
        CloseFiles;
        HALT
      END;
      IF word.CharString[0]=')' THEN
        IF write THEN
          FWriteString(MODfile,word.CharString) END;
        EXIT END
      ELSE
        GetWord(File,word) END
    END
  END
END CommentHandler;

PROCEDURE QuoteHandler(File: FileNum; write: BOOLEAN; Qtype: CHAR);
(* Reads quoted strings (even if nested) from File and writes
them to the MODfile if write=TRUE. Qtype is a character
indicating which type of quote (single or double) to check
for. *)
VAR
  word : Word;
BEGIN
  GetWord(File,word);
LOOP
  IF word.CharString[word.FirstChar+word.Length]=EOF THEN
    (* error! *)
    WriteLn;
    WriteString("**** Unexpected end of file found ****");
    WriteLn;
    WriteString(" ** while reading a quoted string **");
    WriteLn;
    CloseFiles;
    HALT
  END;
  IF write THEN FWriteString(MODfile,word.CharString) END;
  (* check to see if this is the end of the quote *)
  IF word.CharString[word.FirstChar+word.Length]=Qtype THEN
    EXIT
  ELSE GetWord(File,word) END
END (* LOOP *)
END QuoteHandler;

PROCEDURE WriteErrorHalt(string1,string2,string3,string4 : ARRAY OF CHAR); (* This routine is used to handle terminal errors found by the monitor pre-processor, usually involving end of file errors. *)
BEGIN
  Write(BELL);
  WriteLn;
  WriteString(string1);
  WriteLn;
  WriteString(string2);
  WriteString(string3);
  WriteString(string4);
  WriteLn;
  CloseFiles;
  HALT
END WriteErrorHalt;

BEGIN
  paren := FALSE
END MLibrary.
DEFINITION MODULE MGetDefs;
(* written by Phil Rosine
  22 Mar 84
  
  This is a part of the MONITR pre-processor. It contains
  routines which get the export list from the DEF file and
  check to see which export items are procedures. It is
  interfaced through the GetDefsMain procedure. *)

(************************************************************************************
 ** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # 
 ** Copyright 1984. Philip E. Rosine #* 
 ** *
 ** The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and *
 ** all parts of the Monitor Pre-processor program. #*
 ** *
 ** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # )

FROM MData IMPORT NameList;

EXPORT QUALIFIED GetDefsMain;

PROCEDURE GetDefsMain(VAR ExpProcList : NameList);

END MGetDefs.
IMPLEMENTATION MODULE MGetDefs;

(*    written by Phil Rosine
     22 Mar 84

This is a part of the MONITR pre-processor. It contains
routines which get the export list from the DEF file and
check to see which export items are procedures. It is
interfaced through the GetDefsMain procedure. *)

(*----------------------------------------------------------
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # *
** Copyright 1984. Philip E. Rosine  *
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # **
** The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and all parts of the Monitor Pre-processor program. **
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # **
**----------------------------------------------------------*)

FROM SYSTEM IMPORT TSIZE;
FROM MData IMPORT NameList, Name, DEFfile, Word, ParseToken;
FROM MFileHandler IMPORT GetWord, CloseFiles;
FROM MLibrary IMPORT Token, CommentHandler, WriteErrorHalt;
FROM StrLibrary IMPORT strcmp, strncmp, strncpy;
FROM Storage IMPORT ALLOCATE;

CONST
    EOF = OC; (* end of file marker *)
    (* the following are error messages to be passed to the WriteErrorHalt procedure *)
    eofDEF = "*** Unexpected end of DEF file found ***";
    expstmt = " ** looking for EXPORT statement **";

VAR
    word : Word; (* for reading from DEFfile *)
    ExpList : NameList; (* global for module *)
    name : NameList; (* used by procedures for allocating memory *)
    next : NameList; (* used as a temp pointer *)

PROCEDURE GetDefsMain(VAR ExpProcList : NameList);
    (* tested 23 Mar 84 -- per -- works OK *)
BEGIN
    ReadExpList;
    FindProcedures(ExpProcList)
END GetDefsMain;
PROCEDURE ReadExpList;
(* finds and reads the EXPORT list from the DEFfile and places it in ExpList *)
(* tested 23 Mar 84 -- per -- works OK *)
BEGIN
  LOOP
    GetWord(DEFFile,word);
    IF word.CharString[word.FirstChar+word.Length]=EOF THEN
      WriteErrorHalt(eofDEF,expstmt,"","")
    END;
    CASE Token(word) OF
      export : EXIT
      I comment : CommentHandler(DEFFile,FALSE)
      ELSE (* nothing *)
    END
    END
  END
  GetWord(DEFFile,word);
  IF word.CharString[word.FirstChar+word.Length]=EOF THEN
    WriteErrorHalt(eofDEF,expstmt,"","")
  END;
  WHILE word.Length=0 DO (* check for comments *)
    IF Token(word)=comment THEN CommentHandler(DEFFile,FALSE) END;
    GetWord(DEFFile,word);
    IF word.CharString[word.FirstChar+word.Length]=EOF THEN
      WriteErrorHalt(eofDEF,expstmt,"","")
    END
  END; (* WHILE *)
  IF word.Length=9 THEN (* check for QUALIFIED *)
    IF strncmp(word.CharString,"QUALIFIED",word.FirstChar,0,9)
    THEN GetWord(DEFFile,word) END
  END
  LOOP
    IF word.CharString[word.FirstChar+word.Length]=EOF THEN
      WriteErrorHalt(eofDEF,expstmt,"","")
    END;
    (* assume no comments within the export statement *)
    ALLOCATE(name,TSIZE(Name));
    IF ExpList=NIL THEN ExpList := name
    ELSE next^.Next := name END;
    next := name;
    strncpy(word.CharString,next^.NameString,
      word.FirstChar,0,word.Length);
    next^.NameString[word.Length] := 0C;
    next^.Length := word.Length;
    next^.Next := NIL;
    IF word.CharString[word.FirstChar+word.Length]=";"
    THEN EXIT
    ELSE GetWord(DEFFile,word) END
PROCEDURE FindProcedures(VAR ExpProcList : NameList);
(* checks which names in ExpList refer to procedures and
return them in ExpProcList *)
(* tested 23 Mar 84 — per — works OK *)
VAR
found : BOOLEAN; (* these are used to search ExpList *)
search, save : NameList;
BEGIN
ExpProcList := NIL;
LOOP
GetWord(DEFFile, word);
CASE Token(word) OF
  procedure :
    (* find procedure name *)
    GetWord(DEFFile, word);
    IF word.CharString[word.FirstChar+word.Length]=EOF THEN
      WriteErrorHalt(eofDEF,"","",""
    END;
    WHILE word.Length=0 DO (* check for comments *)
      IF Token(word)=comment THEN
        CommentHandler(DEFFile, FALSE) END;
      GetWord(DEFFile, word);
      IF word.CharString[word.FirstChar+word.Length]=EOF THEN
        WriteErrorHalt(eofDEF,"","",""
      END
    END; (* WHILE *)
    WHILE Token(word)=comment DO
      CommentHandler(DEFFile, FALSE);
      GetWord(DEFFile, word)
    END;
    (* check the name against the export list *)
    search := ExpList;
    save := ExpList;
    found := FALSE;
    WHILE (NOT found) & (search#NIL) DO
      IF word.Length=search^.Length THEN
        found := strncmp(word.CharString, search^.NameString,
                         word.FirstChar, 0, word.Length)
      END;
      IF NOT found THEN
        save := search; (* remember the previous name *)
      END
    END;
    IF found THEN (* search points to an exported
procedure name *)
    IF ExpProcList=NIL THEN ExpProcList := search
    ELSE next^.Next := search END;
    next := search;
    IF save=ExpList THEN ExpList := next^.Next
    ELSE save^.Next := next^.Next END;
    next^.Next := NIL
END

| comment : CommentHandler(DEFfile,FALSE)
ELSE (* nothing *)
END;
IF word.CharString[word.FirstChar+word.Length]=OC
   THEN (* end of file *) EXIT END

END FindProcedures;

BEGIN
    ExpList := NIL
END MGetDefs.
DEFINITION MODULE MConvert;

(* written by Phil Rosine
24 Mar 84

This is a part of the MONITR pre-processor.
MConvert does the actual conversion of the
MON file into a MOD file by copying the MON
file and adding the statements and declarations
required. It is interfaced through the
ConvertMain procedure. *)

(***************************************************************************
**                        Copyright 1984. Philip E. Rosine                      **
**                        **
** The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and **
** all parts of the Monitor Pre-processor program. **
**                        **
***************************************************************************)

FROM MData IMPORT NameList;

EXPORT QUALIFIED ConvertMain;

PROCEDURE ConvertMain(VAR ExpProcList : NameList);

END MConvert.
IMPLEMENTATION MODULE MConvert;

(* written by Phil Rosine
24 Mar 84

This is a part of the MONITR pre-processor. MConvert
does the actual conversion of the MON file into a MOD
file by copying the MON file and adding the statements
and declarations required. It is interfaced through
the ConvertMain procedure.*)

(* ******************************************************
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # *
** Copyright 1984. Philip E. Rosine
**
** The copyright covers both DEFINITION and IMPLI-
** CEMENTATION MODULES of PreemptiveScheduler, and
** all parts of the Monitor Pre-processor program.
**
** # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # *
** ******************************************************)

FROM SYSTEM IMPORT TSIZE;
FROM Storage IMPORT ALLOCATE;
FROM MData IMPORT NameList, Name, MONfile, MODfile, DATfile,
   Word, ParseToken;
FROM MFileHandler IMPORT GetWord, CloseFiles;
FROM MLibrary IMPORT Token, CommentHandler, QuoteHandler,
   WriteErrorHalt;
FROM StrLibrary IMPORT strcmp, strncmp, strncpy;
FROM FInOut IMPORT FRead, FReadString,
    FWtite, FWtiteString, FWtiteLn;

CONST
    StringSize = 80;
    size = 3; /* number of lines for Entry and Exit statements */
    Dquote = 42C; /* double quote */
    Squote = 47C; /* single quote */
    EOF = 0C; /* end of file marker */
    /* the following are error messages to be passed to the
      WriteErrorHalt procedure */
    eofMON = "*** Unexpected end of MON file found ***";
    endofProc = " ** looking for end of procedure ";
    stars = " **";
    expProc = " ** looking for exported procedure name **";
    mutexadd = " ** while adding mutual exclusion in PROCEDURE ";
    nameProc = " ** looking for procedure name **";
    noProc = "*** No PROCEDURE declarations ***";
mainBEGIN = " ** looking for BEGIN main module **";
vardec = " ** reading variable declarations **";

TYPE
String = ARRAY [0..StringSize] OF CHAR;
State = (statel, state2, state3, state4, state5);
AddToken = (import, Type, mutex, cond, entry, exit, init);

VAR
CondList : NameList;
Entry, Exit : ARRAY [1..size] OF String;

PROCEDURE ConvertMain(VAR ExpProcList : NameList);
(* tested with dummy AddMonitor - 26 Mar 84 - per - tested OK *)
(* tested - 2 Apr 84 - per - tested OK *)
VAR
state : State;
word : Word;

PROCEDURE ConstHandler;
BEGIN
AddMonitor(import);
FWriteString(MODfile, word.CharString);
state := state2
END ConstHandler;

PROCEDURE TypeHandler;
BEGIN
IF state=statel THEN AddMonitor(import) END;
FWriteString(MODfile, word.CharString);
AddMonitor(Type);
state := state3
END TypeHandler;

PROCEDURE VarHandler;
BEGIN
IF state=statel THEN AddMonitor(import) END;
IF state#state3 THEN
  FWriteString(MODfile, "TYPE"); FWriteLn(MODfile);
  AddMonitor(Type)
END;
FWriteString(MODfile, word.CharString);
AddMonitor(mutex);
state := state4
END VarHandler;

PROCEDURE ProcedureHandler;
BEGIN
IF state=statel THEN AddMonitor(import) END;
IF (state=state1) OR (state=state2) THEN
    FWriteString(MODfile,"TYPE"); FWriteLn(MODfile);
    AddMonitor(Type)
END;
IF state#state4 THEN
    FWriteString(MODfile,"VAR"); FWriteLn(MODfile);
    AddMonitor(mutex)
END;
AddMonitor(cond); (* add procedure declarations *)
state := state5
END ProcedureHandler;

PROCEDURE MutexHandler;
(* This procedure handles adding mutual exclusion to exported procedures. *)
VAR
    found : BOOLEAN; (* these are used to search ExpProcList *)
    search' : NameList;
    saveword : Word;

    (* The local procedure ProcedureCopy is used by MutexHandler to process nested procedure declarations and to copy local procedures (not exported by the monitor and therefore not requiring mutual exclusion. *)
PROCEDURE ProcedureCopy;
VAR
    PName : Word;
BEGIN
    (* the procedure name has already been found and written out to the MONfile. *)
    PName := word; (* remember the procedure name *)
    GetWord(MONfile,word);
    LOOP
        IF word.CharString[word.FirstChar+word.Length]=EOF THEN
            WriteErrorHalt(eofMON,endofProc,PName.CharString,stars)
        END;
        CASE Token(word) OF
            end :
                FWriteString(MODfile,word.CharString);
                IF word.CharString[word.FirstChar+word.Length]#¨;` (* `END;` NOT found *)
                THEN (* this may be the end of the procedure *)
                    GetWord(MONfile,word);
                    IF word.CharString[word.FirstChar+word.Length]=EOF
                        THEN
                            WriteErrorHalt(eofMON,endofProc,
                                PName.CharString,stars)
                        END;
            END;
        WHILE (word.Length=0) &
            (word.CharString[word.FirstChar+word.Length]#¨|¨)
DO (* check for comments *)
    IF Token(word)=comment
        THEN CommentHandler(MONfile,TRUE)
        ELSE FWriteString(MODfile,word.CharString)
    END;
    GetWord(MONfile,word);
    IF word.CharString[word.FirstChar+word.Length]=EOF
        THEN
            WriteErrorHalt(eofMON,endofProc,
                            PName.CharString,stars)
    END
END; (* WHILE *)
    IF (word.Length=PName.Length) 
      & strncmp(word.CharString,PName.CharString,
                    word.FirstChar,PName.FirstChar,word.Length)
        THEN (* end of procedure being copied *)
            FWriteString(MODfile,word.CharString);
            EXIT
        END
    ELSE GetWord(MONfile,word)
    END
    | comment :
        CommentHandler(MONfile,TRUE);
        GetWord(MONfile,word)
    ELSE
        FWriteString(MODfile,word.CharString);
        IF (word.CharString[word.FirstChar+word.Length]=Squote)
            OR (word.CharString[word.FirstChar+word.Length]=Dquote)
            THEN
                QuoteHandler(MONfile,TRUE,
                               word.CharString[word.FirstChar+word.Length])
            END;
            GetWord(MONfile,word)
        END (* CASE *)
    END (* LOOP *)
END ProcedureCopy;

BEGIN (* Mutexhandler *)
    FWriteString(MODfile,word.CharString);
    GetWord(MONfile,word);
    IF word.CharString[word.FirstChar+word.Length]=EOF
        THEN
            WriteErrorHalt(eofMON,expProc,"",""
        END;
    WHILE word.Length=0 DO (* check for comments *)
        IF Token(word)=comment
            THEN CommentHandler(MONfile,TRUE)
            ELSE FWriteString(MODfile,word.CharString)
            END;
        GetWord(MONfile,word);
        IF word.CharString[word.FirstChar+word.Length]=EOF
            THEN
                WriteErrorHalt(eofMON,expProc,"",""
            END
        END
May 28 15:26 1984

pgms/mconve.mod Page 5

A - 88

END; (* WHILE *)
FWriteString(MODfile,word.CharString);
(* check the name against the export list *)
search :** ExpProcList;
found := FALSE;
WHILE (NOT found) & (search#NIL) DO
IF word.Length=search".Length THEN
found ':*= strncmp(word.CharString,search''.NameString,
word.FirstChar,0,word.Length)
END;
IF NOT found THEN search := search".Next END
END;
IF found THEN
(* search points to an exported
procedure name *)
GetWord(MONfile,word);
LOOP
IF word.CharStringtword.FirstChar+word.Lengthl^EOF THEN
WriteErrorHalt(eofMON,mutexadd,search".Name Str i n g ,stars)
END;
CASE Token(word) OF
begin :
FWriteString(MODfile,word.CharString);
AddMonitor(entry);
GetWord(MONfile,word)
I return :
AddMonitor(exit);
FWriteString(MODfile,word.CharString);
GetWord(MONfile,word)
I end :
IF word.CharString[word.FirstChar+word.Length]#';'
(* 'END;' NOT found *)
THEN (* this may be the end of the procedure *)
saveword := word;
GetWord(MONfile,word);
IF w o r d .Char Str ing[wor d.Fir stChar+word.Leng t h ]=E0F
THEN
WriteErrorHalt(eofMON,endofProc,
search".NameString,stars)
END;
WHILE (word.Length=0) &
(word.CharString[word.FirstChar+word.Length]#'I')
DO (* check for comments *)
IF Token (word ^ c o m m e n t
THEN CommentHandler(MONfile,TRUE)
ELSE FWriteString(MODfile,word.CharString)
END;
GetWord(MONfile,word);
IF word.CharString[word.FirstChar+word.Length]=E0F
THEN
WriteErrorHalt(eofMON,endofProc,


search^,NameString,stars)

END

END; (* WHILE *)

IF (word.Length=search^,Length) &
    strncmp(word.CharString,search^,NameString,
    word.FirstChar,0,word.Length)
THEN (* end of exported procedure *)
    AddMonitor(exit);
    (* NOTE: saveword contains the END
     statement and word contains the
     procedure name. If there was a
     comment between them, it is now out of
     place before the exit statement. *)
    FWriteString(MODfile,saveword.CharString);
    FWriteString(MODfile,word.CharString);
    EXIT
ELSE (* end of something else *)
    FWriteString(MODfile,saveword.CharString);
    (* since the ';' could have followed a
     comment *)
    FWrite(MODfile,';'); (* just in case ... *)
    IF Token(word)#end THEN
        FWriteString(MODfile,word.CharString);
        GetWord(MONfile,word)
    END
END

ELSE (* in this case the END statement is in word *)
    FWriteString(MODfile,word.CharString);
    GetWord(MONfile,word)
END

procedure : (* nested procedure *)
    FWriteString(MODfile,word.CharString);
    GetWord(MONfile,word);
    IF word.CharString[word.FirstChar+word.Length]=EOF
    THEN
        WriteErrorHalt(eofMON,nameProc," ","")
    END;
    WHILE word.Length=0 DO (* check for comments *)
        IF Token(word)=comment
            THEN CommentHandler(MONfile,TRUE)
            ELSE FWriteString(MODfile,word.CharString) END;
            GetWord(MONfile,word);
            IF word.CharString[word.FirstChar+word.Length]=EOF
            THEN
                WriteErrorHalt(eofMON,nameProc," ","")
            END
        END
    END; (* WHILE *)
    (* procedure name is now in word *)
    FWriteString(MODfile,word.CharString);
    ProcedureCopy;
GetWord(MONfile, word)
| comment :
| CommentHandler(MONfile, TRUE);
| GetWord(MONfile, word)
ELSE
| FWriteString(MODfile, word.CharString);
| IF (word.CharString[word.FirstChar + word.Length] = Squote)
| OR (word.CharString[word.FirstChar + word.Length] = Dquote)
| THEN
| QuoteHandler(MONfile, TRUE,
| word.CharString[word.FirstChar + word.Length])

END; GetWord(MONfile, word)
END (* CASE *)
END (* LOOP *)
ELSE (* non-exported procedure found *)
ProcedureCopy
END
END MutexHandler;

BEGIN (* ConvertMain *)
state := state1;
LOOP
| GetWord(MONfile, word);
CASE state OF
| state1, state2, state3 :
| IF word.CharString[word.FirstChar + word.Length] = EOF THEN
| WriteErrorHalt(eofMON, ",", ",", ",")

END;
CASE Token(word) OF
| const : ConstHandler
| type : TypeHandler
| var : VarHandler
| procedure :
| ProcedureHandler;
| MutexHandler
| comment : CommentHandler(MONfile, TRUE)
| begin : (* this is an error! *)
| WriteErrorHalt(noProc, ",", ",", ",")
ELSE
| FWriteString(MODfile, word.CharString);
| IF (word.CharString[word.FirstChar + word.Length] = Squote)
| OR (word.CharString[word.FirstChar + word.Length] = Dquote)
| THEN
| QuoteHandler(MONfile, TRUE,
| word.CharString[word.FirstChar + word.Length])

END (* CASE Token for states 1, 2, or 3 *)
| state4 : (* reached from VarHandler *)
| BuildCondList(word);
PROCEDURE BuildCondList(VAR word : Word);
(* tested 26 mar 84 - per - tested OK *)
VAR
  Temp : ARRAY [0..15] OF Word;
  i, count : CARDINAL;
  name, next : NameList;
BEGIN
  count := 0;
  LOOP
    IF word.CharString[word.FirstChar+word.Length]=EOF THEN
      WriteErrorHalt(eofMON,vardec,"","")
    END;
    CASE Token(word) OF
      procedure : EXIT
      | comment : CommentHandler(MONfile,TRUE)
      ELSE
        FWriteString(MODfile,word.CharString);
        IF (word.CharString[word.FirstChar+word.Length]=Squote)
          OR (word.CharString[word.FirstChar+word.Length]=Dquote)
        THEN
          QuoteHandler(MONfile,TRUE,
                        word.CharString[word.FirstChar+word.Length])
        END;
    END (* CASE Token for state5 *)
  END (* CASE state *)
END (* LOOP *)
END ConvertMain;

ProcedureHandler;
MutexHandler
| state5 : (* reached from ProcedureHandler *)
  IF word.CharString[word.FirstChar+word.Length]=EOF THEN
    WriteErrorHalt(eofMON,mainBEGIN,"","")
  END;
CASE Token(word) OF
  procedure : MutexHandler
  | begin :
    FWriteString(MODfile,word.CharString);
    AddMonitor(init);
    WHILE word.CharString[word.FirstChar+word.Length]#EOF DO
      GetWord(MONfile,word);
      FWriteString(MODfile,word.CharString)
    END;
    EXIT (* end of MONfile *)
  | comment : CommentHandler(MONfile,TRUE)
  ELSE
    FWriteString(MODfile,word.CharString);
    IF (word.CharString[word.FirstChar+word.Length]=Squote)
      OR (word.CharString[word.FirstChar+word.Length]=Dquote)
    THEN
      QuoteHandler(MONfile,TRUE,
                    word.CharString[word.FirstChar+word.Length])
    END;
  END (* CASE Token for state5 *)
END (* LOOP *)

(* word contains a variable name *)
IF word.Length#0 THEN
  Temp[count] := word; INC(count)
END;
IF word.CharString[word.FirstChar+word.Length]=":" THEN
  DEC(count);
  GetWord(MONfile,word);
  IF word.CharString[word.FirstChar+word.Length]=EOF THEN
    WriteErrorHalt(eofMON,vardec,"",""
  END;
END; (* WHILE *)
FWriteString(MODfile,word.CharString);
(* check the type *)
IF (word.Length=9) & strncmp(word.CharString,"CONDITION", word.FirstChar,0,9) THEN
  (* copy the temp list to the CondList *)
  FOR i:=0 TO count DO
    ALLOCATE(name,TSIZE(Name));
    IF CondList=NIL THEN CondList := name
    ELSE next^.Next := name END;
    next := name;
    strncpy(Temp[i].CharString,next^.NameString, Temp[i].FirstChar,0,Temp[i].Length);
    next^.NameString[Temp[i].Length] := 0C;
    next^.Length := Temp[i].Length;
    next^.Next := NIL;
  END (* FOR *)
ELSE (* find end of type designator *)
  WHILE word.CharString[word.FirstChar+word.Length]#";" DO
    GetWord(MONfile,word);
    IF word.CharString[word.FirstChar+word.Length]=EOF THEN
      WriteErrorHalt(eofMON,vardec,"",""
    END;
    IF Token(word)=comment THEN
      CommentHandler(MONfile,TRUE)
    ELSE FWriteString(MODfile,word.CharString) END;
  END
END
END (* WHILE *)
END; (* IF CONDITION *)
count := 0
END (* IF ';' *)
END; (* CASE *)
GetWord(MONfile,word);
END (* LOOP *)
END BuildCondList;

PROCEDURE AddMonitor(token : AddToken);
(* tested -- 2 Apr 84 -- per -- tested OK *)
VAR
  i : CARDINAL;
  string : String;
  ch : CHAR; (* used to read the end of line from the DATfile *)
BEGIN
CASE token OF
  import, Type, mutex :
    FWriteLn(MODfile);
    REPEAT
      FReadString(DATfile,string); FRead(DATfile,ch);
      FWriteString(MODfile,string); FWriteLn(MODfile)
    UNTIL strncmp(string,"(*** end",0,0,7);
  cond :
    FWriteLn(MODfile);
    REPEAT
      FReadString(DATfile,string); FRead(DATfile,ch);
      FWriteString(MODfile,string); FWriteLn(MODfile)
    UNTIL strncmp(string,"(*** end",0,0,7);
    FOR i:=1 TO size DO
      FReadString(DATfile,Entry[i]); FRead(DATfile,ch)
    END;
    FOR i:=1 TO size DO
      FReadString(DATfile,Exit[i]); FRead(DATfile,ch)
    END
  entry :
    FWriteLn(MODfile);
    FOR i:=1 TO size DO
      FWriteString(MODfile,Entry[i]); FWriteLn(MODfile)
    END
  exit :
    FWriteLn(MODfile);
    FOR i:=1 TO size DO
      FWriteString(MODfile,Exit[i]); FWriteLn(MODfile)
    END
  init :
    FWriteLn(MODfile);
    REPEAT
      FReadString(DATfile,string); FRead(DATfile,ch);
fwritestring(modfile,string); fwriteln(modfile)
until strcmp(string,"(*** end",0,0,7);
fwritestring(modfile,"(*** init conditions ***)");
fwriteln(modfile);
while condlist#nil do
  fwritestring(modfile,"initcondition(");
  fwritestring(modfile,condlist^.namestring);
  fwritestring(modfile,";");fwriteln(modfile);
  condlist := condlist^.next
end;
fwritestring(modfile,"(*** end init conditions ***)");
fwriteln(modfile)
end (* case *)
end addmonitor;

begin
  condlist := nil
end mconvert.
(* This module has been processed by the monitor pre-processor MONITR. A number of lines have been added to the code by the pre-processor. These lines are preceded and followed by comment lines with three asterisks like this comment. *)

(* import *)
FROM PreemptiveScheduler IMPORT SEMAPHORE, P, V, initsemaphore;
(* end import *)

(* type CONDITION *)
CONDITION = RECORD
  c :  SEMAPHORE;
  cc :  CARDINAL
END;
(* end type CONDITION *)

(* mutex variable declarations *)
MONITORSEM :  SEMAPHORE;
MONITORCOND :  CONDITION;
(* end mutex variable declarations *)

PROCEDURE WAIT(VAR C :  CONDITION);
BEGIN
  INC(C.cc);
  IF MONITORCOND.cc > 0
  THEN V(MONITORCOND.c)
  ELSE V(MONITORSEM)
  END;
  P(C.c);
  DEC(C.cc)
END WAIT;

(* })
PROCEDURE SIGNAL(VAR C :  CONDITION);
BEGIN
  INC(MONITORCOND.cc);
  IF C.cc > 0
  THEN
    V(C.c);
    P(MONITORCOND.c)
  END;
  DEC(MONITORCOND.cc)
END SIGNAL;
(* })

PROCEDURE initcondition(VAR C :  CONDITION);
BEGIN
  initsemaphore(C.c,0);
  C.cc := 0
END initcondition;
(* end cond procedure declarations *)

(* entry *)
  P(MONITORSEM);
(* end entry *)

(* exit *)
; IF MONITORCOND.cc>0 THEN V(MONITORCOND.c) ELSE V(MONITORSEM) END
(*** end exit ***)
(*** init ***)
initsemaphore(MONITORSEM,1);
initcondition(MONITORCOND);
(*** end init ***)
DEFINITION MODULE MTest;

(* written by Phil Rosine
21 Mar 84

This is a test module to test certain portions of the MONITR pre-processor. *)

(*****************************************************************************
**                                                                            *
**                        Copyright 1984. Philip E. Rosine                       *
**                                                                            *
** The copyright covers both DEFINITION and IMPLEMENTATION MODULES of PreemptiveScheduler, and all parts of the Monitor Pre-processor program.
**                                                                            *
*******************************************************************************)

EXPORT QUALIFIED TestWord, TestToken, TestGetDefs;

PROCEDURE TestWord;
  (* to test the GetWord procedure *)

PROCEDURE TestToken;
  (* to test the Token procedure *)

PROCEDURE TestGetDefs;
  (* to test the GetDefs module *)

END MTest.
IMPLEMENTATION MODULE MTest;

(* written by Phil Rosine
21 Mar 84

This is a test module to test certain portions of the MONITR pre-processor. *)

(*--------------------------------------------------------------
************'*****************
*# #* 
*# Copyright 1984. Philip E. Rosine *#
*# 
*# The copyright covers both DEFINITION and IMPLEMENTATION
*# MODULES of PreemptiveScheduler, and
*# all parts of the Monitor Pre-processor program. *
*# 
*# *******************************************************
******************************************************)

FROM InOut IMPORT Write, WriteString, WriteLn, WriteCard;
FROM FInOut IMPORT FWriteString;
FROM MFileHandler IMPORT GetWord;
FROM MLibrary IMPORT Token, CommentHandler;
FROM MData IMPORT Word, DEFfile, MONfile, MODfile,
ParseToken, NameList;
FROM MGetDefs IMPORT GetDefsMain;

VAR
i : CARDINAL;
word : Word;

PROCEDURE TestWord;
(* to test the GetWord procedure *)
BEGIN
WriteLn;WriteString("DEF File:");WriteLn;
i := 1;
REPEAT
GetWord(DEFfile,word);
Write('/
');
WriteString(word.CharString);
Write('/
');
WriteCard(word.FirstChar,3);
Write('/
');
WriteCard(word.Length,3);
Write('/
');
INC(i)
UNTIL (i>25) OR
(word.CharString[word.FirstChar+word.Length]=0C); WriteLn;WriteString("MON File:”);WriteLn;
i := 1;
REPEAT
  GetWord(MONfile,word);
  Write("/”);
  WriteString(word.CharString);
  Write("/”);
  WriteCard(word.FirstChar,3);
  Write("/”);
  WriteCard(word.Length,3);
  Write("/”);
  INC(i)
UNTIL (i>25) OR (word.CharString[word.FirstChar+word.Length]=0C);
END TestWord;

PROCEDURE TestToken;
  (* to test the Token function *)
  (* This was also used to test the CommentHandler routine. When this was done
   the FWriteString(MODfile...) statements were added. This test correctly copied
   the .MON file into the .MOD file. 23 Mar 84 - per *)
BEGIN
  WriteLn;WriteString("DEF File:");WriteLn;
  REPEAT
    GetWord(DEFfile,word);
    CASE Token(word) OF
      procedure :
        WriteString("procedure/”);
        WriteString(word.CharString);
        WriteLn
      comment :
        CommentHandler(DEFfile,FALSE);
        WriteString("comment/”);
        WriteString(word.CharString);
        WriteLn
      export :
        WriteString("export/”);
        WriteString(word.CharString);
        WriteLn
      begin :
        WriteString("begin/”);
        WriteString(word.CharString);
        WriteLn
      const :
        WriteString("const/”);
        WriteString(word.CharString);
        WriteLn
    END;
  END;
END TestToken;
| type : |
  | WriteString("type/"); |
  | WriteString(word.CharString); |
  | WriteLn |
| var : |
  | WriteString("var/"); |
  | WriteString(word.CharString); |
  | WriteLn |
| end : |
  | WriteString("end/"); |
  | WriteString(word.CharString); |
  | WriteLn |
| other : |
| ELSE |
  | WriteString("error/"); |
  | WriteString(word.CharString); |
  | WriteLn |
END; 
UNTIL (word.CharString[word.FirstChar+word.Length]=0C); 

WriteLn;WriteString("MON File:");WriteLn;
REPEAT 
GetWord(MONfile,word); 
CASE Token(word) OF 
  | procedure : 
    | FWriteString(MODfile,word.CharString); |
    | WriteString("procedure/"); |
    | WriteString(word.CharString); |
    | WriteLn |
  | comment : 
    | CommentHandler(MONfile,TRUE); |
    | WriteString("comment/"); |
    | WriteString(word.CharString); |
    | WriteLn |
  | export : 
    | FWriteString(MODfile,word.CharString); |
    | WriteString("export/"); |
    | WriteString(word.CharString); |
    | WriteLn |
  | begin : 
    | FWriteString(MODfile,word.CharString); |
    | WriteString("begin/"); |
    | WriteString(word.CharString); |
    | WriteLn |
  | const : 
    | FWriteString(MODfile,word.CharString); |
    | WriteString("const/"); |
    | WriteString(word.CharString); |
    | WriteLn |
  | type :
PROCEDURE TestGetDefs;
(* to test the GetDefs module *)
VAR
  list : NameList;
BEGIN
  GetDefsMain(list);
  WHILE list#NIL DO
    WriteString(list^.NameString);WriteLn;
    list := list^.Next
  END;
END TestGetDefs;

BEGIN
  (* nothing *)
END MTest.
Appendix B -- User's Guides
1. Overview

Four modules have been provided to handle I/O functions from within Modula-2 programs. Two of these, InOut and RealInOut, are specified by Wirth [1983]. These two are implemented almost exactly as Wirth has specified, with the differences noted below. The third module, FINOut, is essentially a combination/extension of InOut and RealInOut designed to have all I/O done through files. The fourth, IOControl, is provided to handle interaction between concurrent processes which require I/O. The DEFINITION MODULEs are listed at the end of this guide.

2. The Modules

2.1. InOut

This module contains procedures which allow reading and writing of data from and to the standard input and output devices. In general, this module follows the usage described by Wirth, and further explained by the comments in the DEFINITION MODULE. The delete key will work for corrections during input from the terminal.
2.1.1. **Files** The standard I/O devices are, by default, the user's terminal. The procedures OpenInput and OpenOutput allow a program to change the standard devices to be files. The name of a file is obtained by prompting the user, and then reading a filename and an optional extension from the terminal. With module InOut, a program cannot open a file without prompting the user for the name, which is a slight inconvenience. Module FInOut is provided to solve this problem. The procedure call requires a character string constant or variable (three characters long) which will define the default file extension. If the user enters an escape character as the file name, the terminal will be used instead of a file. If an existing file is opened for output, the old file is renamed with a 'Q' as the first character of the extension (*.Q*); any file with that name (*.Q*) is lost. This protective backup is not required by Wirth. Once a file is opened for input or output it remains open until the program ends or the file is closed. In order to ensure data integrity, calls to CloseInput and CloseOutput should be used if the corresponding Open procedures have been used. When a file is closed, the device is reassigned to be the terminal, and another call to Open can assign another file. Correct file closure upon program termination is not guaranteed if the file is not explicitly closed.

2.1.2. **String I/O** InOut handles string I/O slightly differently than Wirth specifies. A string, the way Wirth uses it for ReadString and WriteString, is really just a single word: no spaces are allowed. Module InOut, as implemented, handles true strings. ReadString will discard all leading blanks and control characters, and then read all characters until it encounters some non-printing character (control character), a tab, a linefeed, or a carriage return. This definition of a string is the same as that allowed by Modula-2 for quoted string constants. The string returned to the calling procedure will be terminated by a null (OC) character; if more characters are input than are allowed in the variable, they will be lost. Procedure WriteString will output any character until it finds a null (OC) character or the maximum length of the variable has been output. Strings read in by ReadString, and quoted string constants meet this criteria; strings built by user program modules should be null terminated to prevent output errors.
2.1.3. Numeric I/O

INTEGER and CARDINAL number I/O is handled by InOut in accordance with Wirth's specifications. All numeric output procedures require an output field width, and the field will be printed as '*'s if it is not wide enough. As noted in the comments, hexadecimal and octal output always require 4 or 6 (respectively) spaces for output at a minimum (otherwise the field is '*' filled). Base ten output requires 1 to 6 spaces, depending upon the expected range of the variable. When numbers are being read, spaces, tabs, and control characters (including linefeed, etc.) terminate the input, and thus cannot be corrected by use of the delete key.

2.1.4. Completion codes and termination characters

Module InOut exports two variables, 'Done' and 'termCH'. 'Done' is a BOOLEAN which indicates the success or failure of a read operation. The exact definition of its value is dependent upon which read procedure was last called; the DEFINITION module gives further explanation for each procedure. The read operations always read one more character than they return (except 'Read', which reads only a single character). This extra character, which terminates the read operation, is NOT pushed back on the input stream, the way it is in many languages. Instead it is placed in the variable 'termCH' for the program to check and use as desired. This is an important consideration. For instance, if a program is to read and copy a file by using ReadString and WriteString, ReadString will terminate on a newline, a tab, or the end of file (assuming no odd control characters). If a tab or newline is in 'termCH', the program must explicitly output the tab by a call Write(llB) or a newline by calling WriteLn. 'termCH' equal to ASCII null (0B) indicates an end of file when reading from disk.

2.2. RealInOut

This module handles reading and writing real numbers in a manner similar to InOut. RealInOut, in fact, uses exactly the same devices and files as used by InOut, and use of OpenInput (etc) from InOut affects the use of RealInOut in a corresponding manner. Reading of reals is done by reading a string, so spaces will be included in the input string, but will terminate the actual computation of the real value from the string. The effect
of this is that a real number stored in a file must be isolated from other data by control characters (usually by putting each one on a separate line, although tabs could be used); this is not true of integer and cardinal numbers, which can be separated by spaces. Octal output of reals is done in machine format, i.e., two octal words are printed, and the user must interpret them. A fieldwidth must be specified for decimal real output. For octal output, no fieldwidth is used; octal reals always take up 13 character spaces on output.

Module RealInOut exports a return code variable 'Done' in the same way InOut does. It does not export a termination character; the character terminating a call to ReadReal is lost. This appears to be a bug in the specification which has not been corrected by this implementation.

2.3. **FINOut**

This module is provided to allow use of multiple files by a program, and to allow opening of files without having to prompt the user. All of the facilities provided by InOut and RealInOut are provided in exactly the same manner by FINOut, except that FINOut will not print any error messages on the user's terminal. Instead, the programmer must check FtermCH and FDone to insure completion of the desired action and handle errors accordingly.

2.3.1. **Use of FINOut** The procedure and variable names are identical to those of the other modules except that they have an 'F' at the beginning of the name, and the first parameter of the procedure is the channel number. The channel number can be any CARDINAL on the range 1..15, giving access to 15 separate files at any one time. These 15 files include any files (not the terminal) opened by calls to InOut. In general, if both InOut and FINOut are to be used, InOut should be used only for terminal I/O and all file I/O should be handled by FINOut.

2.3.2. **File Names** When a file is opened, a file name is required. This must be in RT11 form, i.e.: DDDDDDDDDFEEEE where DDD is the device, FFFFFF is the file name, and EEEE is the file extension. All 12 characters must be supplied in the form shown, without
Therefore, if the desired file is DK:TEST.DAT (a 2 char device and a 4 char name) the filename sent to FInOut must be specified as the string 'DK TEST DAT'. There is no provision within FInOut for getting a user supplied filename. Module FileNames provides a procedure for reading a filename from the terminal, and InOut can be used to prompt the user. The following program shows how it can be done:

```modula-2
MODULE demo;
(* written by Phil Rosine
   Mar 84 *)
FROM FInOut IMPORT FDone,FOpenOutput,FCloseOutput,
     WriteString,FWriteLn;
FROM InOut IMPORT WriteString,WriteLn,Read;
FROM FileNames IMPORT ReadFileName;
FROM Files IMPORT FileName;
CONST
    A = 1;
VAR
    fileA : FileName;
    ch : CHAR;
BEGIN
    REPEAT
        WriteString("Output file ==> ");
        ReadFileName(fileA,'DK OUT DAT');
        Read(ch);WriteLn;
        (* filename 'DK OUT DAT' (DK:OUT.DAT)
          will be returned by default if the user
          enters a null line (a carriage return).
          If only a name (say 'TFILE') is entered,
          then DK:TFILE.DAT will be returned,
          etc. The Read(ch) purges the input
          stream of the CR since the filename
          was terminated by the linefeed. If this
          is not done, an error causing a loop
          based on FDone will read a null filename
          on the next try and open DK:OUT.DAT. *)
        FOpenOutput(A,fileA);
    UNTIL FDone;
    FWriteString(A,"test");FWriteLn(A);
    FCloseOutput(A)
END demo.
```
2.4. IOControl

Since concurrent processes using these I/O modules must share the same devices, some care must be taken to insure data integrity. One means of doing this is to have a single process, possibly with a buffer, handle all I/O for the program, but this is not always feasible. A second way is to ensure that processes desiring I/O operations acquire absolute access to the device or procedure before doing I/O. A program can do this by using semaphores in conjunction with the scheduling process (see the manual on module PreemptiveScheduler). A module, IOControl, has been provided to do this for the programmer. Two procedures, AcquireIO and ReleaseIO, are exported by this module. In order for it to work correctly, all processes using InOut or RealInOut must use these IOControl procedures rigorously.

AcquireIO insures that a process gets private access to the I/O functions. If some other process already has control of I/O, then the current process is halted until access to I/O is available. ReleaseIO allows another process to acquire the I/O. These procedures must always be used in pairs, and the call to AcquireIO must always precede the call to ReleaseIO. Failure to follow these rules will cause unpredictable results, and possible interference between processes or deadlock. See the notes in the DEFINITION MODULE IOControl.

REFERENCES

[Wirth, 1983].
DEFINITION MODULE InOut; (* N.Wirth 19.2.80*)

(* modified by Philip E. Rosine
17 Nov 83

modification was done to make this module
compatible with the definition for InOut
given in [Wirth,1983]. *)

(* General Notes:

Several of these operations are refer to any
character less than or equal to a blank; this is
denoted as <=" " in the comments. Since the
system uses ASCII, characters <=" " include all
control characters, tab, carriage return, and line
feed, as well as spaces. *)

FROM SYSTEM IMPORT WORD;
EXPORT QUALIFIED
EOL, Done, termCH,
OpenInput, OpenOutput, CloseInput, CloseOutput,
Read, ReadString, ReadInt, ReadCard,
Write, WriteLn, WriteString, WriteInt, WriteCard,
WriteOct, WriteHex;

CONST EOL = 15C; (* this is a carriage return *)

VAR
Done : BOOLEAN;
(* Done is used to indicate the success or failure
of a procedure call to InOut. The exact meaning
is dependent upon which call is made...
see the notes for each procedure. *)
termCH : CHAR;
(* termCH will contain the last character read by
any of the read operations; ie, the character
which terminated the read, not the last character
in data read. This character is NOT pushed back
on the input stream, and if it is needed by the
next data item, the calling program must add it to
the data itself. The initial value of termCH is
not specified. *)

PROCEDURE OpenInput(defext:ARRAY OF CHAR);
(* request a file name and open file "in".
Done := "file was successfully opened".
If open, subsequent input is read from this file.

May 23, 1984 p 104 Philip E. Rosine
If name does not have an extension, add extension defext. Default device is DK:, default filename is IN. If <ESC> is input, TTY: will become the input.

PROCEDURE OpenOutput(defext:ARRAY OF CHAR);
(* request a file name and open output file "out". Done := "file was successfully opened".
If open, subsequent output is written on this file. If name does not have an extension, add extension defext. Default device is DK:, default filename is OUT. If <ESC> is input, TTY: will become the output. *)

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 in.eof *)

PROCEDURE ReadString(VAR s:ARRAY OF CHAR);
(* read string, ie, sequence of characters not containing control characters, tabs, carriage return, or linefeed. Input is terminated by any character < " "; this character is assigned to termCH. DEL is used for backspacing when input from terminal. The last character of the string s is OC. If more characters are input than the size of s would allow for, they will be lost. Done is TRUE if the number of characters read is >0. *)

PROCEDURE ReadInt(VAR x: INTEGER);
(*skip blanks and control characters; if a sequence of digits (possibly preceded by a sign) follows, read it as a decimal integer. Done indicates whether a number was read, termCH gives the last character read, i.e. the one following the sequence of digits. No tests for overflow are made. DEL is used to backspace for terminal input. Input is terminated by any character <= " ". *)

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

May 23, 1984 p 105 Philip E. Rosine
Leading Blanks are ignored. Done := "cardinal was read". DEL is used to backspace for terminal input. Input is terminated by any character <= " ". For both ReadCard and ReadInt, if the input data includes non-numeric data before the termCH (<=" "), the numeric data up to that point will be converted and placed in the parameter variable, and the following data (up to termCH) will be lost. Done will be FALSE in this case, although the data in x may be valid. Some data from the input stream will have been lost. *)

PROCEDURE Write(ch: CHAR);
PROCEDURE WriteLn; (* terminate line with carriage return followed by linefeed *)
PROCEDURE WriteString(s: ARRAY OF CHAR);
  (* The string s is assumed to be any string of characters up to either the maximum size of the ARRAY s or until the first null (0C) is found. A quoted string constant meets these standards. The compiler will not allow quoted strings to contain characters < " " . Use Write with an octal or hex constant parameter to output control characters, tabs, or form control characters. *)
PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
  (* write integer x with (at most) n characters on file "out" . If n is greater than the number of digits needed, blanks are added preceding the number. If the number requires more than n characters, the field will be filled with '*'. *)
PROCEDURE WriteCard(x, n: CARDINAL);
PROCEDURE WriteOct(w, n: CARDINAL);
  (* n must be >=6 *)
PROCEDURE WriteHex(w, n: CARDINAL);
  (* n must be >=4 *)

END InOut.

May 23, 1984 p 106 Philip E. Rosine
DEFINITION MODULE RealInOut;
(* as defined by Wirth, 1983, pp 104-5
entered by Philip E. Rosine
25 Nov 83 *)

EXPORT QUALIFIED ReadReal, WriteReal, WriteRealOct, Done;

VAR Done: BOOLEAN;

PROCEDURE ReadReal(VAR x: REAL);
(* Read REAL number x according to syntax:
["+"|"-"|digit{digit}]["."digit{digit}]["E"["+"|"-"|digit{digit}]
Done := "a number was read".
At most 7 digits are significant, leading zeroes not counting. Maximum exponent is 38. Input terminates with a blank or any control character. DEL is used for backspacing. *)

PROCEDURE WriteReal(x: REAL; n: CARDINAL);
(* Write x using n characters. If fewer than n characters are needed, leading blanks are inserted. *)

PROCEDURE WriteRealOct(x: REAL);
(* Write x in octal form -- two octal words are written showing the memory representation of the real x -- fieldwidth is 13 *)

END RealInOut.
DEFINITION MODULE FInOut;
(* modified by Philip E. Rosine
27 feb 84
This is a modification of InOut in order to allow
opening of multiple files for input or output. *)

IMPORT Files;

EXPORT QUALIFIED
FEOL, FDone, FtermCH, ChanNum,
FOpenInput, FOpenOutput, FCloseInput, FCloseOutput,
FRead, FReadString, FReadInt, FReadCard,
FWrite, FWriteLn, FWriteString, FWriteInt, FWriteCard,
FWriteOct, FWriteHex, FReadReal, FWriteReal,
FWriteRealOct;

CONST FEOL = 15C;     (* this is a carriage return *)

TYPE ChanNum = [1..15];    (* channel numbers *)

VAR
FDone : BOOLEAN;
(* FDone is used to indicate the success or
failure of a procedure call to InOut. The exact
meaning is dependent upon which call is made...
see the notes for each procedure. *)
FtermCH : CHAR;
(* FtermCH will contain the last character read by
any of the read operations; ie, the character
which terminated the read, not the last character
in data read. This character is NOT pushed back
on the input stream, and if it is needed by the
next data item, the calling program must add it to
the data itself. The initial value of FtermCH is
not specified. *)
(* It is more or less assumed that the programmer
can remember whether a channel is open for input
or output, and read/write appropriately. However,
in the case of a read operation to a write channel,
FtermCH is assigned OC (end-of-file) and FDone will
be FALSE. For a write operation to a read channel,
FDone will be assigned FALSE. These can be checked
if necessary. The critical error checks for the
programmer are to insure that no more than 15 files
are open at any given time. This includes any files
opened by calls to InOut.OpenInput or InOut.Open-
Output. FDone will be returned as FALSE if this happens, but the system will not print an error message. Error messages for this case are the programmer's responsibility. The same result will be obtained from trying to open file which is already open (which could happen due to excessive implementation hiding in IMPORTed modules). *)

PROCEDURE FOpenInput(InFno:ChanNum;InFN:Files.FileName);
(* open file InFN for input on channel InFno.*
FDone := "file was successfully opened".
If open, subsequent input is read from this file.
*)

PROCEDURE FOpenOutput(OutFno:ChanNum;OutFN:Files.FileName);
(* open output file OutFN on channel OutFno.*
FDone := "file was successfully opened".
If open, subsequent output is written on this file. *)

PROCEDURE FCloseInput(InFno:ChanNum);
(* closes input file *)

PROCEDURE FCloseOutput(OutFno:ChanNum);
(* closes output file *)

PROCEDURE FRead(Fno:ChanNum;VAR ch: CHAR);
(* FDone := NOT in.eof *)

PROCEDURE FReadString(Fno:ChanNum;VAR s:ARRAY OF CHAR);
(* read string, ie, sequence of characters not containing control characters, tabs, carriage return, or linefeed. Input is terminated by any character < " " ; this character is assigned to FtermCH. The last character of the string s is 0C. If more characters are input than the size of s would allow for, they will be lost. FDone is TRUE if the number of characters read is >0. *)

PROCEDURE FReadInt(Fno:ChanNum;VAR x: INTEGER);
(*skip blanks and control characters; if a sequence of digits (possibly preceded by a sign) follows, read it as a decimal integer. FDone indicates whether a number was read, termCH gives the last character read, i.e. the one following the sequence of digits. No tests for overflow are made. Input
is terminated by any character <= " ". *)

PROCEDURE FReadCard(Fno:ChanNum;VAR x: CARDINAL);
(* read string and convert to cardinal. Syntax:
   cardinal=digit{digit}.
Leading Blanks are ignored. FDone := "cardinal
was read". Input is terminated by any character <=
" ". For both ReadCard and ReadInt, if the input
data includes non-numeric data before the FtermCH
(<=" "), the numeric data up to that point will be
converted and placed in the parameter variable,
and the following data (up to FtermCH) will be lost.
FDone will be FALSE in this case, although the data
in x may be valid. Some data from the input stream
will have been lost. *)

PROCEDURE FWrite(Fno:ChanNum;ch: CHAR);

PROCEDURE FWriteLn(Fno:ChanNum);
(* terminate line with carriage return followed by
   linefeed *)

PROCEDURE FWriteString(Fno:ChanNum;s: ARRAY OF CHAR);
(* The string s is assumed to be any string of
   characters up to either the maximum size of the
   ARRAY s or until the first null (0C) is found. A
   quoted string constant meets these standards. The
   compiler will not allow quoted strings to contain
   characters < " ". Use Write with an octal or hex
   constant parameter to output control characters,
   tabs, or form control characters. *)

PROCEDURE FWriteInt(Fno:ChanNum;x: INTEGER; n: CARDINAL);
(* write integer x with (at most) n characters on
   file "out". If n is greater than the number of
digits needed, blanks are added preceding the
number. If the number requires more than n
characters, the field will be filled with '*'. *
)

PROCEDURE FWriteCard(Fno:ChanNum;x, n: CARDINAL);
(* n must be >=6 *)

PROCEDURE FWriteOct(Fno:ChanNum;w, n: CARDINAL);
(* n must be >=4 *)

PROCEDURE FWriteHex(Fno:ChanNum;w, n: CARDINAL);
PROCEDURE FReadReal(Fno:ChanNum;VAR x: REAL);
  (* Read REAL number x according to syntax:
   [+|-][digit{digit}][.][digit{digit}][E[+|-][digit{digit}]]
   FDone := "a number was read".
   At most 7 digits are significant, leading zeroes
   not counting. Maximum exponent is 38. Input
   terminates with a blank or any control character. *)

PROCEDURE FWriteReal(Fno:ChanNum;x: REAL; n: CARDINAL);
  (* Write x using n characters. If fewer than n
   characters are needed, leading blanks are inserted. *)

PROCEDURE FWriteRealOct(Fno:ChanNum;x: REAL);
  (* Write x in octal form -- two octal words are
   written showing the memory representation of the
   real x -- fieldwidth is 13 *)

END FInOut.
DEFINITION MODULE IOControl;

(* written by Phil Rosine
Jan 84

This module provides the means to control access
to I/O functions when running concurrent processes. *)

EXPORT QUALIFIED AcquireIO, ReleaseIO;

PROCEDURE AcquireIO;
(* This procedure guarantees the user process
absolute access to the I/O functions so protected,
provided that all user processes call AcquireIO
prior to using I/O. It cannot guarantee against
improper use. *)

PROCEDURE ReleaseIO;
(* This releases I/O which was previously acquired. *)

(* USAGE EXAMPLE ::

PI:

... AcquireIO;
WriteString("test PI");
WriteLn;
ReleaseIO;
...

P2:

... AcquireIO;
WriteCard(P2^..ID);
WriteLn;
ReleaseIO;
...

*)

END IOControl.
1. Overview

Modula-2, in addition to the procedure (sub-routine) calls which are common to most high-level languages, also has facilities for using co-routines and for writing program specific device interrupt traps. The TRANSFER statement allows a process to transfer control unconditionally to another process; the IOTRANSFER statement allows the process to set an interrupt vector which will transfer control to the desired process when an interrupt is received. By writing an interrupt routine for the system clock interrupt, a programmer can develop a program which acts as a process scheduler, inter-leaving processes running on a single CPU.

There are some difficulties with doing this, particularly with doing it for every program written. In order to overcome these difficulties, a module has been developed which allows a programmer to simulate concurrent process execution, and control the interactions between the processes. Using the module PreemptiveScheduler, a program can start and stop subprocesses, control synchronization between processes, and provide for mutual
exclusion of common data areas and procedures. This module is similar to (and based upon) the module Processes described by Wirth [1983]. There are, however, major differences between this module and that described by Wirth.

Wirth's Processes module is simply an extension of standard Modula-2 which makes it easier for a programmer to use co-routines. It does not set up true scheduling of processes over time using the clock interrupt facility. On the other hand, the PreemptiveScheduler module sets up the kernel of time-sharing operating system where concurrent processes started by a program share the CPU over time. This implies an indeterminacy of process interaction which is lacking in Wirth's module. Additional procedures and data structures are provided to allow the programmer/designer to deal with this indeterminate nature of the resulting program. The rest of this guide will discuss the procedures and data structures provided, and their intended use. Possible problems will also be considered, and the checks built into PreemptiveScheduler will be noted, as well as the errors which the module cannot protect against.

2. Exported Procedures and Data Structures

A concurrent processing system requires two related sets of functions: 1) the capability of starting and stopping independently executing processes, and 2) some means of controlling the interaction of these processes. For lack of better terms, the first of these will be called Process Control functions, and the second will be called Process Interaction functions. In addition to these two basic sets of functions, a trace feature is included with the PreemptiveScheduler module.

2.1. Process Control

As mentioned above, PreemptiveScheduler constitutes the kernel, or central control unit, of a time-sharing operating system. It contains, however, some features which make it different from the average operating system kernel. In fact, the module was not designed to be an operating system; it was designed to be used for instruction and experimentation in concurrent programming. As such, PreemptiveScheduler, rather than being itself the base process of the system like an operating
system kernal, is a part of a program which itself is running under a single-user (non-time-sharing) operating system. As an experimental tool, PreemptiveScheduler must take control of the operating system during program execution, and then relinquish control at the end. Most operating systems, in fact, never halt once they start, except due to physical interference (turning the machine off). PreemptiveScheduler could be used in this way; however, it also contains facilities which allow it to halt.

This halting requirement of a program running in an experimental environment is the prime difficulty with the IOTRANSFER system call of Modula-2. IOTRANSFER does not necessarily let go of an interrupt trap once it finishes, and this causes problems in returning control to the normal operating system. PreemptiveScheduler contains procedure calls which allow a program to start time sharing processes, and to halt them, either individually or all at once. Certain checks are built into the system, since if the scheduler is not correctly halted it will prevent the operating system from functioning after the program ends. In order to allow maximum flexibility (ie, allow a true operating system kernal to be built), it is possible to write programs, intentionally or in error, which will take control of the machine and not release it.

There are two process control procedures: start and HaltScheduler. Two other procedure functions are related to process control: StatusOfScheduler and ProcessID. These are all explained in detail in the following sections. The discussion should be read carefully in order to prevent errors and the necessity of re-booting the whole machine.

2.1.1. Starting Processes - PROCEDURE start

The procedure 'start' initializes a procedure as a process and places it in the ready queue. The first call to 'start' also initializes the clock interrupt so that time-sharing occurs among the ready processes. The main program is also considered to be a process running concurrently with any processes started. This is not to imply that only the program module can call 'start'. Any procedure may start a process. Obviously, if the first call is in a procedure called by the program module's main line of code, it will eventually return
there; thus the first call to 'start' is considered to be from the 'main' program.

'Start' requires three parameters. The first of these is the name of a procedure of type PROC. Type PROC implies a parameterless procedure. If a process to be started would normally require parameters to be passed, these must be passed via global variables; the procedure definition cannot involve a parameter list. The other two parameters required by 'start' define the workspace for the process. This will ordinarily be an ARRAY OF WORD; WORD is a type which must be imported from module SYSTEM. The parameters passed to 'start' are the address and size of the workspace, which can be obtained by use of the procedures ADR and SIZE imported from SYSTEM. An example is given below. The size of the workspace is of some concern, since if it is too small a run-time error will occur. In general, a size of 100 to 200 words should be enough. The relation of this size to program and procedure size is not clear. The workspace is used (internally by Modula) as a process stack for local variables, parameter lists, etc., and adding new statements to the program can have variable effects on the size required. Statements and procedure calls in one part of the program may have side effects causing procedure calls, etc., in other parts of the program thus requiring increases in workspace size for apparently non-related processes. A fairly safe rule seems to be use 100 or 120 WORDS initially, and increase it as required. The tradeoff is that too large a workspace wastes memory, and if it is too small, the program will not run. (See the discussion of the example program.)

2.1.2. Halting Processes - PROCEDURE HaltScheduler

Since PreemptiveScheduler takes over the system clock interrupt, a procedure is provided to insure correct process and program termination. If the clock interrupt is not correctly reset to the system interrupt vector a non-recoverable system crash may occur. The main program module must not be allowed to end without calling procedure 'HaltScheduler'. If all started processes (procedures of type PROC) reach a normal end before the main program terminates, HaltScheduler will be invoked internally by the scheduler and does not need to be called explicitly by the main. A call to 'HaltScheduler' will halt all processes and return
control to the main line of code, no matter where it is called from. On the other hand, once all concurrent processes (except the main) have been halted by reaching normal procedure end, an automatic call to 'HaltScheduler' is made, returning control to the main, even though the main may be in a wait state.

No parameters are required for 'HaltScheduler'. As a general rule, 'HaltScheduler' will only be called by the main program code, usually as the last thing it does. There are no problems if 'HaltScheduler' is called more than once; in fact, this may be the case whenever the indeterminacy of the concurrent process interaction prevents the programmer from being able to prove that all processes will end prior to the end of the main program. If any started process is 'infinite', a call to 'HaltScheduler' will be mandatory prior to the end of the main program.

2.1.3. Other -- functions StatusOfScheduler and Process

statusID

2.1.3.1. StatusOfScheduler -- The scheduler traps several possible user errors, but no error messages are output, since having PreemptiveScheduler do I/O is likely to cause program errors. Instead, a function procedure, StatusOfScheduler(), is provided. By testing the return value of this function, the user program can determine if errors or unexpected program termination has occurred, and output its own error message. Note that the empty parentheses () are required on the parameterless function call. The return values are as follows:

-1 indicates that the scheduler is running normally. The clock interrupt process has been started and has not been terminated.

0 indicates that the scheduler clock interrupt has not been started.

1 indicates that the scheduler was started and has been halted due to all started processes being halted (by reaching their normal end point, which made an automatic call to HaltScheduler).

2 indicates that the scheduler was started and has been halted by an explicit call to HaltScheduler.
3 indicates an error condition halt of the scheduler. If all processes (including the main) are in a wait state (due to some combination of wait(SIGNAL), P(SEMAPHORE), and normal procedure end), the clock interrupt will recognize that no process is available to run, and therefore no process will ever release one of the waiting processes. In this case, it will call HaltScheduler and shut itself down. Control will be returned to the main program.

2.1.3.2. ProcessID -- When a process is started by a call to 'start', it is assigned a unique CARDINAL number (the main program will always have ID 0). This ID can be obtained by calling ProcessID(), and the return value used to identify which process is currently running.

2.2. Process Interaction

Concurrent processes may interact with each other in a variety of ways for a variety of reasons. In general, however, the two main types of interaction are process synchronization, and the use of shared data areas. Two data types, and operations on them, are provided by PreemptiveScheduler. The types are SIGNAL, which is used for synchronization, and SEMAPHORE, which can be used for both mutual exclusion and synchronization.

2.2.1. Semaphores

Semaphores are the more general of the two data types provided. Type SEMAPHORE is a full-range counting semaphore which can utilize the full range of type CARDINAL for counting purposes. It cannot, however, be manipulated as a CARDINAL. Three operations are provided for use with type SEMAPHORE: P(s), V(s), and initsemaphore(s,n), where 's' is a SEMAPHORE variable, and 'n' is a CARDINAL number (variable or constant).

2.2.1.1. initsemaphore(s,n) -- Before a SEMAPHORE 's' is used, it must be initialized. This is done by calling 'initsemaphore', which assigns an initial value 'n' to the semaphore. The initial value, in a sense, determines the use to which the SEMAPHORE will be put. If 'n' is zero, then the SEMAPHORE will (probably) be used as a synchronization variable; if 'n' is some value greater than zero, then it will be
used for mutual exclusion, with 'n' processes allowed to access the critical region at one time. Use of a SEMAPHORE without initialization has undefined results, and may result in a system crash.

2.2.1.2. P(s) — The P(s) operation is also known as 'wait(s)' by some authors. Dykstra's original terms P and V have been retained for SEMAPHORE operations in this package because 'signal' and 'wait' are used for operation on type SIGNAL. The P(s) operation is defined:

IF s=0 THEN wait (until s is released by a call to V(s) by some process)
ELSE decrement s and proceed.

The P operation thus insures that the calling process will proceed only if fewer than 'n' processes have called P(s) previously.

2.2.1.3. V(s) — The V(s) operation is also known as 'signal(s)' by some authors. Its action is defined as:

IF some process waiting for s THEN release that process
ELSE increment s and proceed.

2.2.2. Signals

Type SIGNAL is less general than the SEMAPHORE, but the operations on it have some qualities making it more valuable than SEMAPHOREs for some synchronization requirements. This SIGNAL is the same as that described by Wirth [1983]. It is nearly identical to the SIGNAL type and operations used within Monitors [Ben-Ari, 1982]. It is important to note, however, that PreemptiveScheduler does not implement a Monitor as a programming primitive. Four operations are supplied for type SIGNAL: wait(s), signal(s), waited(s), and initsignal(s), where 's' is a variable of type SIGNAL.

2.2.2.1. initsignal(s) — Before a SIGNAL 's' may be used, it must be initialized by a call to 'initsignal'. The only parameter is the SIGNAL itself. Use of a SIGNAL without initialization has undefined results, and may result in a system crash.
2.2.2.2. wait(s) -- The wait(s) operation is similar to P(semaphore), except that a wait always occurs. Its action is to suspend operation of the current process until it is released by a call to signal(s) by some other process. Note that for both the P operation and the wait operation, no wait occurs if the scheduler clock is not running (any case where StatusOfScheduler >= 0).

2.2.2.3. signal(s) -- The signal(s) operation is similar to V(semaphore), except that if there are no waiting processes, nothing is done. If there is a waiting process, it is placed in the ready queue. Note that for both the V(semaphore) and signal(s) the calling process continues and the released process (if any) is allowed to continue concurrently (ie, under time-sharing).

2.2.2.4. waited(s) -- The waited(s) operation is a BOOLEAN function which allows a program to check whether or not a process is waiting to for the SIGNAL 's'. If a process is waiting, waited(s) returns TRUE, otherwise it returns FALSE.

2.3. Trace Feature

In order to allow a programmer to check the process interaction resulting from the use of the scheduler, a trace feature has been included. The external interface to this is the procedure TraceOutput. The PreemptiveScheduler module has two implementations, one located in the file PREEMP, and the other in file TRACES. PREEMP has a dummy version of the trace, and calls to TraceOutput have no action. TRACES has the trace package implemented, and TraceOutput creates a file containing data which shows the action of the scheduler. The same DEFINITION module is used for both of these, and the programmer can decide which to use by using the query function of the linker (see usage note below).

2.3.1. Procedure TraceOutput

This procedure needs two parameters, a channel number for the file to be written, and a filename. Module FInOut is used to write the output file, and the I/O user's guide contains more information about channel numbers and filenames. Basically, the channel number
is a CARDINAL in the range 1..15. A filename is a 12 character string of the form DDDNNNNNNNEEE where DDD is the device designator, NNNNNN is the filename, and EEE is the file extension. Any existing file with this name will be replaced.

The trace feature maintains a list of the last 300 accesses to the PreemptiveScheduler. This list is maintained as a circular buffer; the buffer is cleared each time TraceOutput is called. Thus the list will include either the last 300 accesses to the scheduler or all accesses since the last call to TraceOutput.

An access is any call to procedures start, P, V, signal, wait, and HaltScheduler, plus process endings and process changes due to clock interrupts. The data maintained is output in the form:

ProcessID/PC/Action/Variable/NextID

ProcessID is the CARDINAL number assigned to the process by the scheduler (see procedure ProcessID). The PC is the value of the program counter when a clock interrupt occurs. This is output as an octal number. Since it has no meaning except when a clock interrupt occurs, it will be set to 0 on all other Actions. The Action is a code indicating what caused the trace. These codes are as follows:

I = initial entry of process into the process chain
   -- a call to start
C = change of process due to time-slicing
   -- a clock interrupt
H = halted by a call to HaltScheduler
X = halted due to normal procedure end
W = call to wait on a SIGNAL
S = call to signal on a SIGNAL
P = call to wait on a SEMAPHORE
V = call to V on a SEMAPHORE

The Variable is either the address of a procedure (when Action is I) or the address of a SIGNAL (Action is S or W) or SEMAPHORE (Action is P or V). In all other cases Variable is meaningless and is set to 0. NextID is the ProcessID of the next process to be executed (may be the same process). The 300 records will represent about 5 seconds of execution if all of the Actions are of type C.
2.2.2. Using the Trace Feature

Normally the linker will include the standard PreemptiveScheduler module which contains a dummy trace feature. This version (in file PREEMP) is smaller than the TRACES version. A program can use the standard version and contain calls to TraceOutput, but these calls will have no effect. If tracing is desired, the linker must be invoked with the query mode. To do this, add the /q switch to the name of your program when the linker prompts for it. The linker will then pause after each prompt for a module name. Entering a carriage return will cause the standard module to be included. When the PreemptiveScheduler: prompt appears, enter 'traces'. This will cause the trace feature to be linked. Execution of the program will cause the trace file (or files) to be created if and when TraceOutput is called.

2.2.2. Interpreting the Output

The easiest thing to see in the output is the Action. It shows what calls to the scheduler were made. The ProcessID and NextID show what processes were involved. In order to relate the PC and Variable fields to the program, a memory map and source listings will be required. The memory load map is obtained by putting a /m switch on the filename for the linker (in addition to the /q switch). This causes a file with a .MAP extension to be created. The map has the following format:

Storage map of layer producerconsumer (module from file DK:PRODUC.LNK) linked to layer ResidentMonitor from file SY:MODULA.M2S

separate module FilePool from file DK:FilePo.LNK
   Data: 014270B
   Proc # 0: 014272B  Proc # 1: 014316B
   :  similar for each module linked
   :
separate module producerconsumer from file DK:PRODUC.LNK
   Data: 065532B
   Proc # 0: 066516B  Proc # 1: 066652B
   Proc # 2: 066720B
   First free location: 066772B
The starting address for the data area of each module is given, and the starting address for each BEGIN statement in the module. Proc # 0: is the main code for the module, and the other procedures are in the order they are defined in the DEFINITION module (if one exists). Note that this may be different from the order they appear in either the EXPORT statement or the IMPLEMENTATION module. The LST file from the compiler contains offset addresses for each procedure starting at 000000B for the BEGIN statements, and these can be used to locate PC addresses. Data Variables must be located by counting through the data area. Types SIGNAL and SEMAPHORE each require one WORD of storage; type CONDITION (see the Monitor Pre-processor User's Guide) requires two WORDs. Addressing is done by bytes, so you must count each WORD as 2 address locations.

3. Use of PreemptiveScheduler

One very important point: at the beginning of this paper it was noted that all of the scheduler modules mentioned, including PreemptiveScheduler, are designed to 'hide' the system calls TRANSFER, IOTRANSFER, and NEWPROCESS from the programmer. Programs importing PreemptiveScheduler should not use these calls at all. Use of them can have unpredictable effects upon the operation of the scheduler. If a program requires some feature allowed by their use which cannot be obtained by use of the scheduler, then the program designer should work directly with the implementation of the scheduler and produce his own version. The scheduler has been designed so that this will not generally be required, but if it is necessary, a new version of the scheduler will be safer to work with than any attempt to subvert the one supplied.

At the end of this guide is an example program, together with results from two executions, which shows most of the features of the PreemptiveScheduler module. The notes below discuss what is being done, and why, and also point some ways the program could be changed to achieve the same results. The notes refer to the line numbers in the program listing.
3.1. **Program Overview**

This is a relatively straightforward test program. It uses the SEMAPHORE variable '10' to prevent concurrent access of the I/O routines by the processes. After initializing '10' to allow one process at a time, the program starts ten copies of the procedure 'dummy' using workspaces from the ARRAY 'workspace'. These run, executing a dummy loop 1000 times, and printing some trace statements. After all processes are started, the main program waits ten times on SIGNAL variable 'psig' to ensure that all processes end before the main program ends. Each copy of 'dummy' signals 'psig' before it halts. As noted above, if nothing is waiting, the signal(psig) has no effect, and if all processes have halted, the wait(psig) has no effect (execution continues without a wait).

3.2. **Execution Results**

The results of two executions of this program are included. The first has the dummy loop run with the constant 'times' set to 1000 (as shown); the second execution has it set to 60000. The differences are not great, but in the second run, the main program completes and executes its first wait before any process halts. In both runs the interleaving of the processes is apparent. In both cases, the scheduler is actually halted because all the started processes (copies of 'dummy') end, not because the main calls HaltScheduler. In both cases, the majority of waits on 'psig' by the main are 'no-op' calls because there is no running process to signal them.

3.3. **General Notes**

3.3.1. **Workspace Size** — As discussed above, deciding upon the size of the workspace for a process to be started is somewhat difficult. The required size is dependent not only on program size, but upon the actual dynamic execution sequence. In the test program shown, a workspace of 110 WORDs (as in the code) works consistently. A workspace of 90 or 100 WORDs, however, only works if the program output goes to the terminal. If the output is sent to a disk file, the error message:

```
**** Stack overflow
```

is displayed, and the program is halted with a return
to MODULA. If the workspace is 80 WORDs, this error occurs even for terminal output. Using 100 to 120 WORDs seems to be a good rule of thumb. At least 12 WORDs are required by the scheduler, so the program should supply a minimum of 20 WORDs, but it seems doubtful that any reasonable program could get by with such a small workspace. If the stack overflow error occurs, increase the workspace (20 WORDs at a time seems a reasonable amount) until it runs. Rigorous testing should be done to ensure that all the dynamic execution paths are taken. At the very least, testing should cover the worst case, i.e., the case where the longest possible chain of procedure calls occurs. Finding this case may be somewhat difficult, particularly when using system-supplied modules, where the implementation is not known. Extensive testing is the key in these cases.

2.2.2. Program Termination -- As already noted, both inspection of the output and the value of StatusOfScheduler show that the scheduler is halted because all started processes end, not because the main program calls HaltScheduler. It is tempting to eliminate the HaltScheduler call. However, this is dangerous in the present case, since the 'dummy' process also signals the (waiting) main program to resume. It is possible (although not probable) that the processes could signal main in such a way that the main program would end without all processes being able to end. If this happened, the scheduler would not be correctly halted unless the main calls HaltScheduler. This relationship between ending processes and the end of the main program must be considered at all times.

2.2.3. Process Interaction -- This is a related topic to that of program termination. Consider the example program. The main program executes a wait(psig) [line 71] ten times, and each started process 'dummy' executes a signal(psig) [line 41]. Since the signal is followed by the procedure end, it is not really needed. This program will produce the same results if the signal [line 41] is eliminated, and the FOR loop [lines 65 to 72] surrounding the wait [line 71] is also eliminated so that the main executes wait(psig) only once. The wait(psig) is this case forces the main to pause until all the other processes halt. Since we are sure they will all halt, the program will terminate and the main be restarted when the
last process ends. Since this will also halt the scheduler, the call to HaltScheduler [line 83] can also be eliminated. This solution, while it requires less 'overhead' and results in a more compact program, is somewhat tricky and should be used with care.

4. Other Considerations

4.1. I/O

If more than one concurrent processes does input from or output to the same device, some sort of process interaction is likely to result. This can be prevented by allowing only one process to handle all I/O for the program, but that is not always feasible, and may not solve the problem completely unless it is done with a great deal of care. The example program shows how to control this interaction by use of semaphores (the SEMAPHORE IO). A module called IOControl has been provided to assist in handling I/O concurrency when using the modules InOut and RealInOut. It is discussed in the I/O user's guide.

4.2. Module Priority

The use of module priorities as described by Wirth on page 130 of the Modula-2 text [Wirth,1983] does not have quite as general a value as he indicates. Specifically, it does not implement a true monitor. The priority simply prevents an interrupt from stopping execution if the priority of the interrupt is less than or equal to the priority of the executing procedure. This has particular implications for users of PreemptiveScheduler. The module PreemptiveScheduler runs at priority level 6, the same level as the system clock interrupt. This means that it can interrupt any process which is currently running at a lower priority (and therefore force timesharing). On the other hand, a process which is running at priority 6 or higher prevents the interrupt, and is thus not timesharing (ie, no longer running as a concurrent process). Further, a process running (in a module) with priority 6 or 7 which calls a process in another module with priority less than 6 will be interrupted, and thus lose its control of the system. It is apparent that this is not a monitor. If the programmer is certain that his module will not call any other low level module, priority levels 6 and 7 can be used to
temporarily suspend the operation of the scheduler. It should be noted that I/O operations in particular are run at lower priority than the clock interrupt.

REFERENCES

[Ben-Ari, 1982].

[Wirth, 1983].
This is a scheduler module which implements pre-emptive scheduling of processes. Processes are placed in the ready queue by calling start. The scheduling process is initiated by the first call to start. Since the scheduler uses the system clock interrupt vector, the final action of the main program must be a call to HaltScheduler. This restores the original clock interrupt vector and ensures that the operating system will not crash when the program ends. See the notes with procedures start, HaltScheduler, and HaltProcess.

Wirth ["Programming in Modula-2", 1982, pl29ff] discusses the use of implementation module priority levels to implement monitor constructs. As the following discussion demonstrates, this is not precisely true, since setting priorities either will not ensure exclusive access to a module, or will prevent concurrent processing. The implementation module for the scheduler runs at priority level [6], and the hardware clock interrupt also has priority [6]. This means that the clock interrupt will not interrupt calls to the scheduler, and that process interleaving will not take place while any user process is running at level [6] or [7]. This allows monitor effects to be simulated by modules with priority [6] or [7], but at the cost of losing concurrency. A call to such a module will execute until it either finishes with a normal return, until it calls a lower priority module (for instance, and I/O routine), or until it calls wait(signal) or P(semaphore). Since a call to a lower (than [6]) priority module will cause scheduling to resume, the monitor 'effect' is lost, and another process could call the 'monitor' routine successfully even though there is already a process executing it. For this reason, the use of priority levels with this scheduler must be done with extreme care.

If all processes are ever forced into a wait state (this should never happen), the scheduler will force the main program to restart. The function StatusOfScheduler will always return an INTEGER code indicating the current status of the scheduler.

The signal and wait operations on type SIGNAL are primarily...
synchronization primitives for use with the scheduler. The
P and V operations for type SEMAPHORE are primarily for
mutual exclusion in order to control access to critical
resources.
*)
(* modified 12 Apr 84 -- per
Added a trace feature consisting of the procedure
TraceOutput.

There are two IMPLEMENTATION MODULES: one with the trace
feature implemented and one with a dummy trace. See the
note under TraceOutput.
*)

FROM SYSTEM IMPORT ADDRESS;
FROM FInOut IMPORT ChanNum;
FROM Files IMPORT FileName;

EXPORT QUALIFIED wait,waited,signal,start,SIGNAL,ProcessID,
initsignal,HaltProcess,HaltScheduler,
StatusOfScheduler,P,V,SEMAPHORE,initsemaphore,
TraceOutput;

TYPE SIGNAL; (* type SIGNAL is used with the signal and
wait operations to control process
synchronization. signal and wait are the
operations as defined by Hoare for monitors,
but processes under Modula are not monitors. *)
SEMAPHORE; (* type SEMAPHORE is a full-range CARDINAL
valued semaphore as defined by Dykstra. *)
(* both SIGNAL and SEMAPHORE must be initialized
using initsignal or initsemaphore prior to use.
If this is not done, no errors will be generated,
but results are indeterminate. *)

PROCEDURE StatusOfScheduler( ) : INTEGER;
(* indicates current state of the scheduling process:
  0 == never started
-1 == running normally
 1 == halted by HaltProcess
 2 == halted by outside call to HaltScheduler
 3 == halted by error call from Clock -- ie, all
   processes are in a wait state
Errors in the use of the scheduler will not generate
error messages, but will be reflected by
StatusOfScheduler, and can be checked and error
messages generated by the user program. *)

PROCEDURE ProcessID( ) : CARDINAL;
(* This function returns the unique process ID
 number assigned by the scheduler when a process is
 started. The main program will always have ID 0,
 and other processes will have consecutively higher
 numbers in the order they are started. The value
 returned is undefined prior to the first start. *)

PROCEDURE wait(  VAR s: SIGNAL );
(* force process to wait for signal(s) to continue; if
the scheduler clock is not running (ie, no concurrent
processes have been 'start'ed), wait has no action --
it becomes effectively a NO-OP; note that the
scheduler is halted when all concurrent processes
have been halted *)

PROCEDURE waited(  VAR s: SIGNAL ): BOOLEAN;
(* TRUE if a process is waiting on this SIGNAL *)

PROCEDURE signal(  VAR s: SIGNAL );
(* mark process waiting s as ready to proceed; if no
process waiting, then nothing is done *)

PROCEDURE start(  p:PROC; a:ADDRESS; n:CARDINAL );
(* start a process p with workspace a of length n *)
(* 'starting' a process involves placing it in the
ready queue. The first call to start initiates the
clock interrupt process which interleaves the
processes in the ready queue. A started process
should only be exited through a call to HaltProcess.
If started processes are 'infinite', the main
program must call HaltScheduler prior to ending. *)
**Modula-2 Implementation**

DEF MODULE PreemptiveScheduler

PROCEDURE initsignal(VAR s: SIGNAL);
(* initialization of a SIGNAL *)

PROCEDURE HaltScheduler;
(* Halts the scheduling process and restores the
clock interrupt vector... required at the end of
any program which uses the scheduler. *)

PROCEDURE HaltProcess;
(* removes a started process from the ready queue --
if all 'start'ed processes are stopped by calls to
HaltProcess, an automatic call to HaltScheduler is
done by the scheduler and the main program restarted
at whatever point it last was running. *)

PROCEDURE P(VAR s: SEMAPHORE);
(* tests to see if some process is already using s; if
not P acquires access to the region guarded by s,
otherwise pauses until s is released. *)

PROCEDURE V(VAR s: SEMAPHORE);
(* releases s *)

PROCEDURE initsemaphore(VAR s: SEMAPHORE; initvalue: CARDINAL);
(* initializes s to initvalue; initvalue determines how
the SEMAPHORE will work. If it is a positive value n,
then n concurrent process will be allowed to access
whatever region s is protecting. If it is 0, then nothing
will be allowed access to the protected region until a V
operation has been performed. This will usually be used
when synchronization primitive is needed. *)

PROCEDURE TraceOutput(Tfile: ChanNum; filename: FileName);
(* This procedure causes the history of the Preemptive­
Scheduler (for approximately the last 5 seconds) to
be written out to file 'filename'. The trace
history has the form:
ProcessID/PC/Action/Variable/NextID
where ProcessID is the CARDINAL number assigned to the
current process by the scheduler. PC is the address of
the next instruction (ie, the Program Counter). PC only
has validity if the Action code is C, otherwise it will
be zero. Action is a code as follows:
I = initial entry of process into the process chain
C = change of process due to time-slicing
H = halted by a call to HaltScheduler
X = halted due to normal procedure end

May 23, 1984               p 131               Philip E. Rosine
W = caused to wait on a SIGNAL
S = released by a signal on a SIGNAL
P = caused to wait on a SEMAPHORE
V = released by a V on a SEMAPHORE

Variable is the address of the procedure being started as a process if Action is 'I', and is the address of the SIGNAL or SEMAPHORE if Action is 'W', 'S', 'P', or 'V'; in all other cases it will be 0 and has no meaning.

NextID is the number of the next process to be executed by the scheduler (all of these shifts are recorded when the scheduler is trying to switch processes -- ProcessID is the one being switched from and NextID is the one being switched to: they may be the same). NextID will not have any meaning if Action is 'S' or 'V', and will be set to CurrentID in these cases. *)

(* NOTE:
There are two implementation modules for this definition: the file PREEMP.* has a dummy trace feature and procedure TraceOutput has no action. The file TRACES.* has the actual trace feature implemented. To use the package, compile normally, and link with the /q switch if the trace feature is required. When prompted for PreemptiveScheduler, enter 'traces' in order to link in the trace package. *)

END PreemptiveScheduler.
MODULE MultiProcessTest;

(* Written by Phil Rosine
Jan 84

Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADR, SIZE, WORD;
FROM PreemptiveScheduler IMPORT wait, waited, signal,
  SIGNAL, initsignal, start, HaltScheduler, ProcessID,
  P, V, initsemaphore, SEMAPHORE, StatusOfScheduler;
FROM InOut IMPORT WriteString, WriteLn, WriteCard,
  OpenOutput, CloseOutput, WriteInt;

CONST
  numP = 10;
  times = 1000;
  StackSize = 120;

TYPE
  Stack = ARRAY [1..StackSize] OF WORD;

VAR
  i : CARDINAL;
  workspace : ARRAY [1..numP] OF Stack;
  psig : SIGNAL;
  IO : SEMAPHORE;

PROCEDURE dummy;
VAR
  count : CARDINAL;
BEGIN
  P(IO);
  WriteString("dummy loop");
  WriteCard(ProcessID(),3);
  WriteLn;
  V(IO);
  count := 0;
  LOOP
    INC(count);
    IF count>times THEN EXIT END
  END;
  signal(psig);
  P(IO);
  WriteString("dummy loop -- HaltProcess");
  WriteCard(ProcessID(),3);
  WriteLn;
  V(IO);
END dummy;
BEGIN (* MultiProcessTest *)
  initsemaphore(IO,1);
  initsignal(psig);
  P(IO);
  OpenOutput('DTA');
  WriteString("Start MultiP");
  WriteLn;
  V(IO);
  FOR i:=1 TO numP DO
    P(IO);
    WriteString("start dummy");
    WriteCard(i,3);
    WriteLn;
    V(IO);
    start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
  END;
FOR i:=1 TO numP DO
  P(IO);
  WriteString("wait psig");
  WriteCard(i,3);
  WriteLn;
  V(IO);
  wait(psig)
  END;
  P(IO);
  WriteString("StatusOfScheduler = ");
  WriteInt(StatusOfScheduler(),3);
  WriteLn;
  V(IO);
  P(IO);
  WriteString("end MultiP");
  WriteLn;
  CloseOutput;
  V(IO);
  HaltScheduler
END MultiProcessTest.
Modula-2 Implementation Test Program

First Test Run
Loop length = 1000

start ModulP
start dummy 1
start dummy 2
start dummy 3
dummy loop 2
dummy loop 1
start dummy 4
start dummy 5
start dummy 6
dummy loop 6
dummy loop 5
dummy loop 4
dummy loop 3
dummy loop -- HaltProcess 2
dummy loop -- HaltProcess 1
start dummy 7
dummy loop -- HaltProcess 6
dummy loop -- HaltProcess 5
dummy loop -- HaltProcess 4
start dummy 8
dummy loop 7
dummy loop -- HaltProcess 3
start dummy 9
start dummy 10
dummy loop 9
dummy loop -- HaltProcess 7
wait psig 1
dummy loop 10
dummy loop -- HaltProcess 8
dummy loop -- HaltProcess 9
dummy loop -- HaltProcess 10
wait psig 2
wait psig 3
wait psig 4
wait psig 5
wait psig 6
wait psig 7
wait psig 8
wait psig 9
wait psig 10
StatusOfScheduler = 1
end ModulP

May 23, 1984 p 135 Philip E. Rosine
Modula-2 Implementation

Test Program

Second Test Run
Loop length = 60000

start MultiP
start dummy 1
start dummy 2
start dummy 3
start dummy 4
dummy loop 3
dummy loop 2
dummy loop 1
start dummy 5
dummy loop 4
start dummy 6
dummy loop 5
start dummy 7
dummy loop 6
start dummy 8
dummy loop 7
start dummy 9
dummy loop 8
start dummy 10
dummy loop 9
wait psig 1
dummy loop 10
dummy loop -- HaltProcess 3
dummy loop -- HaltProcess 1
wait psig 2
dummy loop -- HaltProcess 4
dummy loop -- HaltProcess 2
dummy loop -- HaltProcess 5
wait psig 3
dummy loop -- HaltProcess 6
dummy loop -- HaltProcess 7
dummy loop -- HaltProcess 8
dummy loop -- HaltProcess 9
dummy loop -- HaltProcess 10
wait psig 4
wait psig 5
wait psig 6
wait psig 7
wait psig 8
wait psig 9
wait psig 10
StatusOfScheduler = 1
end MultiP

May 23, 1984 p 136 Philip E. Rosine
Errata Sheet for Modula-2 Concurrency

Philip E. Rosine

The use of monitors contains one possible error which had not been detected at the time this thesis was finished. In Modula-2, the RETURN statement used in function procedures causes an immediate exit from the procedure and a return to the calling process. The monitor pre-processor has been corrected to insert an exit sequence before any RETURN statement in an EXPORTed function, but the use of functions and RETURNS can cause possible run-time errors.

Since the monitor exit sequence releases mutual exclusion, a waiting process could enter the monitor before the current process finished, since the RETURN must be executed after the exit sequence. This could allow changes to data which would invalidate the action of the original process or the function return value. There appears to be no way around this problem unless monitors are a directly compiled construct within a programming language. Due to this problem, the use of functions as EXPORTed elements of monitors is not recommended.
1. Modula-2 and Monitors

The concept of the monitor as a concurrent programming structure is not directly available in Modula-2. Wirth [1983] discusses the monitor concept briefly, and equates it with the idea of module priority. Unfortunately, priority levels do not quite provide the same control as a true monitor. (For a further discussion, see the PreemptiveScheduler User's Guide.) The program MONITR is provided to allow certain extensions to Modula-2 which allow a programmer to use the monitor as programming structure.

A programmer who desires to use a monitor as a part of his program design writes the monitor as a Modula-2 module, using standard Modula-2 syntax plus certain keyword enhancements. The monitor (DEFINITION and IMPLEMENTATION modules) is then run through the MONITR program. MONITR converts the extensions used to program the monitor into allowable Modula-2 constructs so that the output of MONITR can be processed by the standard Modula compiler and linker. The extensions to the Modula-2 syntax which allow this are discussed below.
2. Value of the Monitor Concept

The PreemptiveScheduler module for Modula-2 provides a programmer with the ability to have a program execute concurrent processes in a time-sharing environment. Semaphores and semaphore operations are provided as a primary means of controlling interaction between concurrent processes. As Ben-Ari [1982] points out, however, semaphores are a rather low-level primitive. In order to provide a more structured method of handling concurrency the monitor concept was developed by Brinch-Hansen [1977] and Hoare [1974].

Semaphores work very well for controlling access to critical regions and for synchronization between processes, but their use relies heavily upon the programmer doing everything correctly and in the right sequence. On the other hand, a monitor, in the simplest case, does not rely on the programmer at all to ensure mutual exclusion. Programming the critical regions as monitors guarantees security against simultaneous access. For synchronization, the monitor SIGNAL and WAIT operations are much simpler to use than P and V operations.

The eventual result of Brinch Hansen's work with monitors was Concurrent Pascal [BrinchHansen,1977], a language which includes monitors as a data type. Adding the monitor as a data type to Modula-2 would be a major and usually unnecessary complication to the Modula concept, but the ability to program modules which can be used in concurrent process environment without having to worry explicitly about mutual exclusion would be a definite advantage. The MONITR program allows the programmer to do this while still using Modula-2 as the program implementation language.

3. The Monitor Preprocessor

3.1. Overview

In order to provide monitors to programmers using Modula-2, a preprocessor has been built. It takes modules written using a single extra data type and two extra keywords (essentially extensions of Modula-2) and converts them into monitors which will work with concurrent programs which use the PreemptiveScheduler module. The programmer only needs to ensure correct
starting and halting of his concurrent processes through the scheduler, and the monitor can be used to handle the process interaction. The terms used by this implementation are those used by Hoare [1974] rather than those used by Brinch Hansen [1977], and their use is discussed in detail in the book by Ben-Ari [1982]. They are:

**TYPE CONDITION**

This is essentially the same as type queue used by Brinch Hansen. It provides a means of defining variables for temporarily delaying processes using the operations SIGNAL and WAIT.

**Operation WAIT(C:CONDITION)**

The WAIT operation causes a process to be queued under a CONDITION variable until it is released by SIGNAL operation on the same CONDITION. Use of a FIFO queue in the implementation ensures fairness for the SIGNAL and WAIT operations. Brinch Hansen used the term 'delay' for the WAIT operation. A WAIT causes the exclusion of other processes to be released in addition to delaying the running process. This means that when the process which has exclusive use of the monitor executes a WAIT(C), another process will be allowed to enter the monitor and run.

**Operation SIGNAL(C:CONDITION)**

The SIGNAL operation releases the next process waiting for CONDITION C, and also causes the running process to wait until the monitor is again available. That is, the waiting process is allowed to execute the monitor where it left off, and the current process is forced to wait until it either exits or again enters a wait state. Whenever control of the monitor is released by a process, either by exiting or by executing a WAIT, processes waiting due to a SIGNAL are given priority to restart over processes waiting for initial entry to the monitor. (See the discussion in Chapter 5 of [Ben-Ari,1982].) Brinch Hansen used 'continue' for the SIGNAL operation.

2.2. *Use of the Preprocessor*
3.2.1. Writing a Monitor

The extra data type provided by use of the preprocessor is CONDITION, and the operators SIGNAL and WAIT are provided for this type. These are used by the programmer as though they were defined by Modula-2, and the preprocessor program takes care of their actual definition.

Monitors are written as normal Modula-2 MODULEs. The DEFINITION MODULE defines what procedures are to be exported, and the preprocessor will convert these procedures into monitor procedures. To ensure correctness, no variables should be exported, but simple data types may be. The pre-processor does not require that no variables be exported, but good programming practice where monitors are used normally requires this restriction. The IMPLEMENTATION module should be written using type CONDITION and the operations SIGNAL and WAIT on that type, just the way Ben-Ari [1982] and others show in their examples. Chapter 5 of Ben-Ari is particularly recommended. CONDITION, SIGNAL, and WAIT are NOT imported: the preprocessor will take care of that. Three keywords, MONITORSEM, MONITORCOND, and initcondition are reserved and must not be used by the program. Otherwise, the syntax, etc., is that of standard Modula-2. The file for the IMPLEMENTATION MODULE should have a .MON extension instead of the normal .MOD extension. Although your main program will use PreemptiveScheduler to start and stop processes, and may use SEMAPHORES and SIGNALs, the monitor module must not do so (results of such possible circumvention of the monitor are uncertain). Note that the keyword SIGNAL is NOT the same as the TYPE of the same name which is exported by PreemptiveScheduler.

One possible error can occur as a result of monitor writing. A procedure within the monitor cannot call any of the exported procedures. This is due to the fact that exported procedures are mutually exclusive, and calling an exported procedure from within the monitor will cause deadlock on the monitor. The scheduler will eventually cause a halt (and return execution to the main program) if this happens, but this occurrence may be hard to debug if the problem is not considered during program design.
3.2.2. Processing the Monitor

The preprocessor is a Modula-2 program run from within the MODULA program (under the RT-11 operating system). This program is named MONITR (the name of the program may be in lowercase, i.e., 'monitr', depending upon your terminal type). When it is run, it will prompt the user for an input file name. The expected (default) file extension for the input file is .MON, but may be different if specified by the user. The output file will have the extension .MOD. If the input file has the .MOD extension, an error message will be output and processing will not occur. A DEFINITION MODULE with the standard .DEF extension must exist for this monitor IMPLEMENTATION MODULE.

Preprocessing consists of adding semaphore operations to ensure mutual exclusion for the exported procedures, and replacement of the CONDITION, SIGNAL, and WAIT keywords with semaphores and semaphore operations. PreemptiveScheduler will be imported. The output file will then be a completely standard Modula-2 module and can be compiled as such. The preprocessor will not find errors in the Modula-2 code, and any compiler errors should be corrected in the .MON monitor file rather than the preprocessor output file (.MOD).

3.2.3. Error Messages

All of these are fatal errors and result in the program halting.

FILE NOT FOUND
If the named input file does not exist on device DK: or the device specified by the user, this message will be output.

NO DEFINITION MODULE FOUND
This message means that no file with the monitor name and a .DEF extension was located on device DK: or the device specified by the user.

INCORRECT INPUT FILE EXTENSION
If the user gives an input filename with a .MOD extension, this error message will be output.

COULD NOT OPEN OUTPUT FILE
If the output file (XXXXXX.MOD) cannot be created, this message will be output. This could happen if the disk is full. It can also happen if the
output file is write-protected. If the file is protected, this message will not occur on the first execution of MONITR; it will occur the second time the program is run. This is because when the output file already exists, it is renamed to be XXXXXXX.QOD, and this can be done even if the file is protected; the second execution tries to delete the back-up file (XXXXXX.QOD) but cannot because of the protection and the file open fails. The XXXXXXX.QOD file will have to be un-protected before the program can be run.

**MONITR DATA FILE NOT FOUND**
The MONITR program requires a data file in order to insert the correct statements to make the monitor into a compilable module. This file is called MONITR.DAT, and if it does not exist on device DK: or SY:, this error will occur. If you cannot find MONITR.DAT, check with the system manager.

*** Unexpected end of DEF file found ***
If the end of the DEFINITION MODULE is found before it is expected, the message will be output, and the program execution will halt. In some cases, a second error message will be output to indicate more about what caused the error; these secondary messages include:

** looking for EXPORT statement **
** reading EXPORT statement **
** looking for procedure name **

*** Unexpected end of MON file found ***
If the end of the IMPLEMENTATION MODULE is found before it is expected, the message will be output, and the program execution will halt. The output file will be produced, and can sometimes be examined to determine exactly what the cause of the error was. In some cases, a second error message will be output to indicate more about what caused the error; these secondary messages include:

** reading variable declarations **
** looking for procedure name **
** looking for exported procedure name **
** looking for end of procedure <name> **
** while adding mutual exclusion in PROCEDURE <name> **
** looking for BEGIN main module **

*** No PROCEDURE declarations ***
A monitor which does not have any procedures can hardly be defined as a monitor, so the preprocessor will interpret this case as an error and halt execution.

2.3. Output File

The output file (XXXXXX.MOD) will have a number of things added in order to make the module work as a monitor. Knowing what these are will make it easier to debug code which does not compile. As noted above, it is not recommended that users try to correct errors in the MOD file, but in order to locate the syntax errors so that they can be corrected in the MON file, a user will have to look at the LST file produced by the compiler, and this will of course be a copy of the MOD file. The line numbers in the LST file will not be of much use since the added statements will throw them off in relation to the MON file.

The statements added are (in order) as follows:

IMPORT PreemptiveScheduler
TYPE CONDITION
VAR MONITORSEM and MONITORCOND
PROCEDURE SIGNAL, WAIT, and initcondition
after the BEGIN for each exported procedure, an entry statement will be added to ensure mutual exclusion
before the END for each exported procedure, an exit statement will be added to ensure mutual exclusion
after the BEGIN for the module, initialization statements will be added for MONITORSEM, MONITORCOND, and all user-declared CONDITION variables

Since the pre-processor does not check syntax, there are some things that it expects which may cause it to fail. One of these is failure to have a main body in the MON file. The pre-processor looks for the BEGIN statement of the module in order to know where to put the initialization statements for CONDITION and mutual exclusion variables. Likewise, if the exported procedures do not have correct end statements (END PROCname), the pre-
Modula-2 Implementation Monitor Preprocessor User's Guide

processor cannot find the place to put the procedure EXIT statement to control mutual exclusion. The MON file also must have the declarations in 'standard' order:

```
IMPLEMENTATION MODULE modulename;
IMPORT ...;
:
IMPORT ...;
CONST ...;
TYPE ...;
VAR ...;
PROCEDURE ...;
:
PROCEDURE ...;
BEGIN
:
END modulename.
```

Modula-2 syntax does not formally require this order, but it seems standard enough that following it should not be a hardship. Not all of these are required by any module, of course, but if they are used, they must be in that order so that MONITR can find things and add what it needs to add.

REFERENCES

[Ben-Ari,1982].

[BrinchHansen,1977].

[Hoare,1974].
Appendix C

Appendix C — Test Programs

The test programs cover a wide range. Some of them are rather minimal, while others are fairly detailed. Some of these were used to test earlier versions of modules than those shown in the source code section (Appendix A). For instance, several of these programs import HaltProcess from PreemptiveScheduler, but HaltProcess is no longer exported by the module. There are several sections of test programs, starting with those which test general features of Modula-2 and the run-time system, and then those which test the specific modules being developed.
The following programs were written to test various functions of Modula-2 on our system. There are several versions of some of these programs.

testglom :: tests module priority interaction
glom :: used by testglom
_tFUnct__:: tests compilation of parameterless functions
Function :: tests definition of parameterless functions
TCoRoutines :: tests coroutine interaction
TClock :: tests clock interrupts
TVArName :: test possible variable naming problems
MultiProcessTest :: tests previous schedulers
MODULE testglom;

(* to test module priority interaction *)
(* tested with both glom and PreemptiveScheduler at
various priorities: *)
glom:: P-Sched:: result

<table>
<thead>
<tr>
<th>glom</th>
<th>P-Sched</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7]</td>
<td>[7]</td>
<td>glom not interrupted</td>
</tr>
<tr>
<td>[6]</td>
<td>[7]</td>
<td>glom not interrupted</td>
</tr>
<tr>
<td>[5]</td>
<td>[7]</td>
<td>glom interrupted</td>
</tr>
<tr>
<td>[4]</td>
<td>[7]</td>
<td>glom interrupted</td>
</tr>
<tr>
<td>[3]</td>
<td>[7]</td>
<td>glom interrupted</td>
</tr>
<tr>
<td>[2]</td>
<td>[7]</td>
<td>glom interrupted</td>
</tr>
<tr>
<td>[1]</td>
<td>[7]</td>
<td>glom interrupted</td>
</tr>
<tr>
<td>none</td>
<td>[7]</td>
<td>glom interrupted</td>
</tr>
</tbody>
</table>

FROM SYSTEM IMPORT ADR, SIZE, WORD;
FROM PreemptiveScheduler IMPORT SIGNAL, wait, start, initsignal, HaltProcess, HaltScheduler;
FROM glom IMPORT bigloop, x;
FROM InOut IMPORT WriteCard, WriteLn;

VAR
  test : SIGNAL;
gWSP, pWSP : ARRAY [0..100] OF WORD;

PROCEDURE print;
  VAR i : CARDINAL;
BEGIN
  FOR i:=0 TO 10 DO
    WriteCard(x,10);WriteLn
  END;
  HaltProcess
END print;

BEGIN
  initsignal(test);
  start(bigloop,ADR(gWSP),SIZE(gWSP));
  start(print,ADR(pWSP),SIZE(pWSP));
  wait(test)
END testglom.
DEFINITION MODULE glom;

    (*   To test priority level interaction.  *)

EXPORT QUALIFIED bigloop, x;

VAR x : CARDINAL;

PROCEDURE bigloop;

END glom.
IMPLEMENTATION MODULE glom[5];

(* To test priority level interaction. *)

FROM PreemptiveScheduler IMPORT HaltProcess;

CONST times = 64000;

PROCEDURE bigloop;
  VAR i: CARDINAL;
  BEGIN
    FOR i := 1 TO times DO x := x + 1 END;
    HaltProcess
  END bigloop;

BEGIN
  x := 0
END glom.
MODULE tFunction;

(* written by Phil Rosine 
   feb 84 

to test compilation of a parameterless function. 

RESULTS: 
A previous program with a similar parameterless function 
causes problems during compilation... this one seems 
to flag the error correctly. A parameterless function 
requires a null parameter list (). *) 

FROM InOut IMPORT WriteCard, WriteLn;

PROCEDURE function: CARDINAL; BEGIN 
   RETURN 0 
END function;

BEGIN 
   WriteCard(function);  WriteLn
END tFunction.
DEFINITION MODULE Function;

(* written by Phil Rosine
   feb 84

to test definition of a parameterless function.

RESULTS:
   Previous attempts to define a parameterless function caused
   compiler problems. This seems to flag the error correctly. *)

EXPORT QUALIFIED function;

PROCEDURE function: CARDINAL;

END Function.
MODULE TCoRoutines;

(* Written by Phil Rosine
14 dec 83

to test coroutine interaction....
*)

(* RESULTS:
works just like it should *)

FROM SYSTEM IMPORT WORD, TRANSFER, PROCESS, NEWPROCESS, ADR, SIZE;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR
    P1, P2 : PROCESS;
    wsp1, wsp2 : ARRAY [0..100] OF WORD;
    p1, p2 : CARDINAL;

PROCEDURE procl;
BEGIN
    WriteString("enter procl");WriteLn;
    NEWPROCESS(proc2,ADR(wsp2),SIZE(wsp2),P2);
    p1 := 0;
    TRANSFER(P1,P2);
    INC(p1);WriteCard(p1,8);
    TRANSFER(P1,P2);
    INC(p1);WriteCard(p1,8);
    TRANSFER(P1,P2);
    INC(p1);WriteCard(p1,8);
    TRANSFER(P1,P2);
    INC(p1);WriteCard(p1,8);
    END procl;

PROCEDURE proc2;
BEGIN
    WriteString("enter proc2");WriteLn;
    p2 := 0;
    TRANSFER(P2,P1);
    INC(p2);WriteCard(p2,8);WriteLn;
    TRANSFER(P2,P1);
    INC(p2);WriteCard(p2,8);WriteLn;
    TRANSFER(P2,P1);
    INC(p2);WriteCard(p2,8);WriteLn;
    TRANSFER(P2,P1)
    END proc2;

BEGIN
    procl
END TCoRoutines.
MODULE T2CoRoutines;

(* Written by Phil Rosine  
  14 dec 83  
  to test coroutine interaction.... *)

(* RESULTS: 
  works just like it should *)

FROM SYSTEM IMPORT WORD, TRANSFER, PROCESS, NEWPROCESS, ADR, SIZE;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR
  main, P1, P2 : PROCESS;
  wsp1, wsp2 : ARRAY [0..100] OF WORD;
  p1, p2 : CARDINAL;

PROCEDURE procl;
BEGIN
  WriteString("enter procl"); WriteLn;
  NEWPROCESS(proc2,ADR(wsp2),SIZE(wsp2),P2);
  p1 := 0;
  TRANSFER(P1,P2);
  INC(p1); WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1); WriteCard(p1,8); 
  TRANSFER(P1,P2);
  INC(p1); WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1); WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1); WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1); WriteCard(p1,8);WriteLn;
  TRANSFER(P1,main)
END procl;

PROCEDURE proc2;
BEGIN
  WriteString("enter proc2"); WriteLn;
  p2 := 0;
  TRANSFER(P2,P1);
  INC(p2);WriteCard(p2,8);WriteLn;
  TRANSFER(P2,P1);
  INC(p2);WriteCard(p2,8);WriteLn;
  TRANSFER(P2,P1);
  INC(p2);WriteCard(p2,8);WriteLn;
  TRANSFER(P2,P1)
END proc2;

BEGIN
  NEWPROCESS(procl,ADR(wspl),SIZE(wspl),P1);
TRANSFER(main,P1)
END T2CoRoutines.
MODULE T3CoRoutines;

(*  Written by Phil Rosine
   14 dec 83

   to test coroutine interaction....
   this is a change from T3CoRoutines : does not transfer
   back to the main  *)

(*  RESULTS:
   This works OK... it ends with the message:
   **** end of coroutine
   and returns to MODULA.  *)

FROM SYSTEM IMPORT WORD, TRANSFER, PROCESS, NEWPROCESS, ADR, SIZE;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR
   main, P1, P2 : PROCESS;
   wsp1, wsp2 : ARRAY [0..100] OF WORD;
   p1, p2 : CARDINAL;

PROCEDURE procl;
BEGIN
  WriteString("enter procl");WriteLn;
  NEWPROCESS(proc2,ADR(wsp2),SIZE(wsp2),P2);
  p1 := 0;
  TRANSFER(P1,P2);
  INC(p1);WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1);WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1);WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1);WriteCard(p1,8);
  TRANSFER(P1,P2);
  INC(p1);WriteCard(p1,8);WriteLn
(*  TRANSFER(P1,main) *)
END procl;

PROCEDURE proc2;
BEGIN
  WriteString("enter proc2");WriteLn;
  p2 := 0;
  TRANSFER(P2,P1);
  INC(p2);WriteCard(p2,8);WriteLn;
  TRANSFER(P2,P1);
  INC(p2);WriteCard(p2,8);WriteLn;
  TRANSFER(P2,P1);
  INC(p2);WriteCard(p2,8);WriteLn;
  TRANSFER(P2,P1)
END proc2;
BEGIN
    NEWPROCESS(procl,ADR(wspl),SIZE(wspl),P1);
    TRANSFER(main,P1)
END T3CoRoutines.
MODULE TClock;

(* written by Phil Rosine
14 dec 83
to test how the clock interrupts work and what the
time interval is *)

(* RESULTS:
Get about 1700 reps of the count loop for each clock
interrupt. Program crashes when it ends, with or
without the TRANSFER back to main. *)

FROM SYSTEM IMPORT WORD, PROCESS, ADDRESS, ADR, SIZE,
NEWPROCESS, TRANSFER, IOTRANSFER;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR
  i,j : CARDINAL;
  clk, P, main : PROCESS;
  cwsp, wsp, mwsp : ARRAY [0..100] OF WORD;

PROCEDURE count;
BEGIN
  LOOP
    INC(i)
  END
END count;

PROCEDURE clock;
BEGIN
  WriteString("enter clock");WriteLn;
  REPEAT
    IOTRANSFER(clk,P,100B);
    INC(j);
    WriteCard(i,8);
    WriteCard(j,8);WriteLn
  UNTIL j>100;
  WriteCard(i,8);
  WriteCard(j,8);WriteLn;
  (* TRANSFER(clk,main) *)
END clock;

BEGIN
  NEWPROCESS(clock,ADR(cwsp),SIZE(cwsp),clk);
  NEWPROCESS(count,ADR(wsp),SIZE(wsp),P);
  i := 0;
  j := 0;
  TRANSFER(main,clk)
END TClock.
MODULE T2Clock;

(* written by Phil Rosine
14 dec 83

to test how the clock interrupts work and what the time interval is *)

(* RESULTS:
Same as for TClock. *)

FROM SYSTEM IMPORT WORD, PROCESS, ADDRESS, ADR, SIZE,
NEWPROCESS, TRANSFER, IOTRANSFER;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR
  i, j : CARDINAL;
  clk, P, main : PROCESS;
  cwsp, wsp, mwsp : ARRAY [0..100] OF WORD;

PROCEDURE count;
BEGIN
  LOOP
    INC(i)
  END
END count;

PROCEDURE clock;
BEGIN
  WriteString("enter clock");WriteLn;
  REPEAT
    IOTRANSFER(clk,P,100B);
    INC(j);
    WriteCard(i,8);
    WriteCard(j,8);WriteLn
  UNTIL j>100;
  WriteCard(i,8);
  WriteCard(j,8);WriteLn
END clock;

BEGIN
  NEWPROCESS(count,ADR(cwsp),SIZE(cwsp),P);
  i := 0;
  j := 0;
  clock
END T2Clock.
MODULE T3Clock;

(* written by Phil Rosine
14 dec 83
to test IOTransfer to the clock *)

(* RESULTS:
Crashes at end, with or without the SYSRESET.
with or without the final WriteString. *)

FROM SYSTEM IMPORT WORD, PROCESS, NEWPROCESS, IOTRANSFER,
ADR, SIZE, TRANSFER, SYSRESET;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR
  P, clk : PROCESS;
  wsp : ARRAY [0..100] OF WORD;
  i : CARDINAL;

PROCEDURE count;
BEGIN
  LOOP
    INC(i)
  END
END count;

BEGIN
  WriteString("start");WriteLn;
  i := 0;
  NEWPROCESS(count,ADR(wsp),SIZE(wsp),P);
  IOTRANSFER(clk,P,100B);
  SYSRESET;
  (* WriteString("end -- count=");WriteCard(i,8);WriteLn *)
END T3Clock.
MODULE T2Clock;

(* written by Phil Rosine
14 dec 83
to test how the clock interrupts work and what the
time interval is *)

(* RESULTS: *)

FROM SYSTEM IMPORT WORD, PROCESS, ADDRESS, ADR, SIZE,
NEWPROCESS, TRANSFER, IOTRANSFER;
FROM InOut IMPORT WriteString, WriteCard, WriteLn;

VAR
  i, j : CARDINAL;
  clk, P, main : PROCESS;
  cwsp, wsp, mwsp : ARRAY [0..100] OF WORD;
  clktrap[100B], clkvect[102B], savtrap, savvect : WORD;

PROCEDURE count;
BEGIN
  LOOP
    INC(i)
  END
END count;

PROCEDURE clock;
BEGIN
  WriteString("enter clock");WriteLn;
  REPEAT
    IOTRANSFER(clk,P,100B);
    clktrap := savtrap; clkvect := savvect;
    INC(j);
    WriteCard(i,8);
    WriteCard(j,8);WriteLn
  UNTIL j>100;
  WriteCard(i,8);
  WriteCard(j,8);WriteLn
END clock;

BEGIN
  savtrap := clktrap; savvect := clkvect;
  NEWPROCESS(count,ADR(cwsp),SIZE(cwsp),P);
  i := 0;
  j := 0;
  clock
END T2Clock.
MODULE TVarName;

(* written by Phil Rosine
Feb 84

to test action of the compiler on similar variable names.

RESULTS:
A similar program with two variables having the first six characters the same did not compile correctly. This one seems to be OK. *)

FROM InOut IMPORT ReadInt, WriteString, WriteInt, WriteLn;

VAR
  VariableOne, VariableTwo : INTEGER;

BEGIN
  WriteString("enter an INTEGER: ");
  ReadInt(VariableOne);
  WriteLn;
  VariableTwo := VariableOne * VariableOne;
  WriteInt(VariableOne, 5); WriteInt(VariableTwo, 5);
  WriteLn
END TVarName.
MODULE MultiProcessTest;

(*    Written by Phil Rosine     
     8 Dec 83                  *)

Designed to test concurrent process interaction.  

FROM SYSTEM IMPORT ADDRESS, ADR, SIZE, PROCESS, WORD;
FROM sched IMPORT wait, waited, signal, SIGNAL,
PAUSE, initsignal, start;
FROM InOut IMPORT WriteString, WriteLn;

CONST
    numP = 1;

TYPE
    Stack = ARRAY [1..100] OF WORD;

VAR
    process: ARRAY [1..10] OF PROCESS;
    i : CARDINAL;
    workspace : ARRAY [1..10] OF Stack;
    psig : SIGNAL;

PROCEDURE dummy;
BEGIN
    LOOP
        WriteString("dummy loop");WriteLn;
        PAUSE(10);
        WriteString("dummy loop -- signaling psig");WriteLn;
        IF waited(psig) THEN signal(psig);
            WriteString("dummy loop -- psig signaled");WriteLn;
    END
END
END dummy;

BEGIN (* MultiProcessTest *)
    WriteString("start MultiP");WriteLn;
initsignal(psig);
    FOR i:=1 TO numP DO
        WriteString("start dummy");WriteLn;
            start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
    END;
    FOR i:=1 TO numP DO
        WriteString("wait psig");WriteLn;
            wait(psig)
    END;
    WriteString("end MultiP");WriteLn
END MultiProcessTest.
MODULE MultiProcessTest;

(* Written by Phil Rosine
8 Dec 83

Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADDRESS, ADR, SIZE, PROCESS, WORD;
FROM scheduler IMPORT wait, waited, signal, SIGNAL,
PAUSE, initsignal, start, HaltProcess, HaltScheduler;
FROM InOut IMPORT WriteString, WriteLn;

CONST
    numP = 1;

TYPE
    Stack = ARRAY [1..100] OF WORD;

VAR
    process: ARRAY [1..10] OF PROCESS;
    i : CARDINAL;
    workspace : ARRAY [1..10] OF Stack;
    psig : SIGNAL;

PROCEDURE dummy;
BEGIN
    LOOP
        WriteString("dummy loop");WriteLn;
        PAUSE(10);
        WriteString("dummy loop — signaling psig");WriteLn;
        IF waited(psig) THEN signal(psig);
        WriteString("dummy loop — psig signaled");WriteLn;
    END
END
dummy;

BEGIN (* MultiProcessTest *)
WriteString("start MultiP");WriteLn;
initsignal(psig);
FOR i:=1 TO numP DO
    WriteString("start dummy");WriteLn;
    start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
END;
FOR i:=1 TO 20000 DO
    (* WriteString("wait psig");WriteLn;
    wait(psig)
    *)
END;
WriteString("halt scheduler");WriteLn;
HaltScheduler;
WriteString("end MultiP");WriteLn
END MultiProcessTest.
The following programs were written to test the modules InOut, RealInOut, and FInOut. There are several versions of some of these programs.

TInOut :: general test program for InOut
TRealInOut :: for RealInOut
MODULE TInOut;
(* written by Philip E. Rosine
27 Nov 83

This program module is designed to test the new version
of InOut as implemented by me, based upon Wirth's 1983
definition and the old implementation (1981). *)

FROM Files IMPORT FileName;
FROM InOut IMPORT
  EOL, Done, termCH,
  OpenInput, OpenOutput, CloseInput, CloseOutput,
  Read, ReadString, ReadInt, ReadCard,
  Write, WriteLn, WriteString, WriteInt, WriteCard,
  WriteOct, WriteHex;

VAR
  x : INTEGER;
  n, y : CARDINAL;
  ch : CHAR;
  CHstring : ARRAY [0..80] OF CHAR;

BEGIN
  LOOP
    WriteString("Enter a String: ");
    ReadString(CHstring);
    WriteString(CHstring); WriteLn;
    IF (CHstring[0] = "E") THEN EXIT END;

    WriteString("Enter a field size: ");
    ReadCard(n); WriteLn;
    WriteString("Enter an integer number: ");
    ReadInt(x); WriteInt(x,n); WriteLn;
    IF Done THEN WriteString("TRUE")
    ELSE WriteString("FALSE") END;WriteLn;
    WriteOct(ORD(termCH),8);WriteLn;
    WriteString("Enter a cardinal number: ");
    ReadCard(y);WriteCard(y,n); WriteLn;
    IF Done THEN WriteString("TRUE")
    ELSE WriteString("FALSE") END;WriteLn;
    WriteOct(ORD(termCH),8);WriteLn;
    WriteInt(x,n); WriteCard(y,n);
    WriteOct(y,n); WriteHex(y,n); WriteLn;

    IF (CHstring[0] = "F") THEN
      OpenOutput("DAT");
      WriteString(CHstring); WriteLn;
      WriteInt(x,n); WriteCard(y,n);
      CloseOutput;
Openlnput("DAT");
ReadString(CHstring);
ReadInt(x);
ReadCard(y);
CloseInput;
WriteLn; WriteString(CHstring); WriteLn;
WriteInt(x,n); WriteCard(y,n); WriteLn;WriteLn;
END
END
END TInOut.
MODULE TinOut;
  (* written by Philip E. Rosine  
      27 Nov 83  *

This program module is designed to test the new version 
of InOut as implemented by me, based upon Wirth's 1983 
definition and the old implementation (1981). *)

FROM Files IMPORT FileName;
FROM InOut IMPORT
  EOL, Done, termCH, 
  OpenInput, OpenOutput, CloseInput, CloseOutput, 
  Read, ReadString, ReadInt, ReadCard, 
  Write, WriteLn, WriteString, WriteInt, WriteCard, 
  WriteOct, WriteHex;

VAR
  x : INTEGER;
  n, y : CARDINAL;
  ch : CHAR;
  CHstring : ARRAY [0..3] OF CHAR;

BEGIN
  LOOP
    WriteString("Enter a String: ");
    ReadString(CHstring);
    WriteString(CHstring); WriteLn;
    IF (CHstring[0] = "E") THEN EXIT END;

    WriteString("Enter a field size: ");
    ReadCard(n); WriteLn;
    WriteString("Enter an integer number: ");
    ReadInt(x); WriteInt(x,n); WriteLn;
    IF Done THEN WriteString("TRUE")
      ELSE WriteString("FALSE") END;WriteLn;
    WriteOct(ORD(termCH),8);WriteLn;
    WriteString("Enter a cardinal number: ");
    ReadCard(y);WriteCard(y,n); WriteLn;
    IF Done THEN WriteString("TRUE")
      ELSE WriteString("FALSE") END;WriteLn;
    WriteOct(ORD(termCH),8);WriteLn;
    WriteInt(x,n); WriteCard(y,n);
    WriteOct(y,n); WriteHex(y,n); WriteLn;

    IF (CHstring[0] = "F") THEN
      OpenOutput("DAT");
      WriteString(CHstring); WriteLn;
      WriteInt(x,n); WriteCard(y,n);
      CloseOutput;
  END LOOP;

END TinOut;
OpenInput("DAT");
ReadString(CHstring);
ReadInt(x);
ReadCard(y);
CloseInput;
WriteLn; WriteString(CHstring); WriteLn;
WriteInt(x,n); WriteCard(y,n); WriteLn;WriteLn;
END

END TInOut.
MODULE TInOut;

    (* written by Phil Rosine
       Jan 84
       to test the size of output files *)

FROM InOut IMPORT OpenOutput, CloseOutput, WriteString, 
               WriteCard, WriteLn;

VAR i : CARDINAL;

BEGIN
  OpenOutput("DAT");
  FOR i:=1 TO 1000 DO
    WriteString("Line number: ");
    WriteCard(i,4);
    WriteLn
  END;
  CloseOutput
END TInOut.
MODULE TInOut;
(* written by Philip E. Rosine  
27 Nov 83  
)

This program module is designed to test the new version of InOut as implemented by me, based upon Wirth's 1983 definition and the old implementation (1981). *)

FROM Files IMPORT FileName;
FROM InOut IMPORT 
  EOL, Done, termCH,
  OpenInput, OpenOutput, CloseInput, CloseOutput,
  Read, ReadString, ReadInt, ReadCard,
  Write, WriteLn, WriteString, WriteInt, WriteCard,
  WriteOct, WriteHex;

VAR 
  x : INTEGER;
  n, y : CARDINAL;
  ch : CHAR;
  CHstring : ARRAY [0..80] OF CHAR;

BEGIN
  LOOP
    WriteString("Enter a String: ");
    ReadString(CHstring);
    WriteString(CHstring); WriteLn;
    IF (CHstring[0] = "E") THEN EXIT END;

    WriteString("Enter a field size: ");
    ReadCard(n); WriteLn;
    WriteString("Enter an integer number: ");
    ReadInt(x); WriteInt(x,n); WriteLn;
    IF Done THEN WriteString("TRUE") 
      ELSE WriteString("FALSE") END;WriteLn;
    WriteOct(ORD(termCH),8);WriteLn;
    WriteString("Enter a cardinal number: ");
    ReadCard(y); WriteCard(y,n); WriteLn;
    IF Done THEN WriteString("TRUE") 
      ELSE WriteString("FALSE") END;WriteLn;
    WriteOct(ORD(termCH),8);WriteLn;
    WriteInt(x,n); WriteCard(y,n);
    WriteOct(y,n); WriteHex(y,n); WriteLn;

    IF (CHstring[0] = "F") THEN
      OpenOutput("DAT");
      WriteString(CHstring); WriteLn;
      WriteInt(x,n); WriteCard(y,n);
      CloseOutput;

Openlnput("DAT");
ReadString(CHstring);
ReadInt(x);
ReadCard(y);
Closelnput;
WriteLn; WriteString(CHstring); WriteLn;
WriteInt(x,n); WriteCard(y,n); WriteLn;WriteLn;
END
END
END TInOut.
MODULE TReallnOut;
(* written by Philip E. Rosine
27 Nov 83

This program module is designed to test the new version
of RealInOut as implemented by me, based upon Wirth's 1983
definition and the old implementation (1981). *)

FROM Files IMPORT FileName;
FROM InOut IMPORT
  Read, ReadString, ReadInt, ReadCard,
  Write, WriteLn, WriteString, WriteInt, WriteCard;
FROM ReallnOut IMPORT
  ReadReal, WriteReal, WriteRealOct, Done;

VAR
  a, b, c, x : REAL;
  n, y : CARDINAL;
  CHstring : ARRAY [0..80] OF CHAR;

BEGIN
  a := 120.0; b := -8.0; c := 1.0; y := 0;
  WriteReal(a,20);WriteLn;
  WriteReal(b,20);WriteLn;
  WriteReal(c,20);WriteLn;

  a := 30000.0;
  b := 100.0;
  c := a*b;
  WriteReal(c,20);WriteLn;
  LOOP
    WriteString("Enter a String ('E' to end): ");
    ReadString(CHstring);
    WriteLn;
    IF (CHstring[0] = "E") THEN EXIT END;

    WriteString("Enter a field size: ");
    ReadCard(n); WriteLn;
    WriteString("Enter an real number: ");
    ReadReal(x); WriteLn;
    WriteReal(x,n);  WriteLn;
    IF Done THEN WriteString("TRUE")
     ELSE WriteString("FALSE") END;WriteLn;
    WriteRealOct(x);WriteLn;
  END
END TReallnOut.
The following programs were written to test the module PreemptiveScheduler. There are several versions of some of these programs.

MultiProcessTest :: tests PreemptiveScheduler
MODULE MultiProcessTest;

(* Written by Phil Rosine
   Jan 84

   Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADR, SIZE, WORD;
FROM PreemptiveScheduler IMPORT wait, waited, signal, SIGNAL,
initsignal, start, HaltScheduler, ProcessID,
P, V, initsemaphore, SEMAPHORE, StatusOfScheduler;
FROM InOut IMPORT WriteString, WriteLn, WriteCard,
       OpenOutput, CloseOutput, WriteInt;

CONST
   numP = 10;
   times = 1000;
   StackSize = 120;

TYPE
   Stack = ARRAY [1..StackSize] OF WORD;

VAR
   i : CARDINAL;
   workspace : ARRAY [1..numP] OF Stack;
   psig : SIGNAL;
   IO : SEMAPHORE;

PROCEDURE dummy;
   VAR
      count: CARDINAL;
   BEGIN
      P(IO);
      WriteString("dummy loop");
      WriteCard(ProcessID(),3);
      WriteLn;
      V(IO);
      count := 0;
      LOOP
         INC(count);
         IF count>times THEN EXIT END
      END;
      signal(psig);
      P(IO);
      WriteString("dummy loop -- HaltProcess");
      WriteCard(ProcessID(),3);
      WriteLn;
      V(IO);
   END dummy;

BEGIN (* MultiProcessTest *)
   initsemaphore(IO,1);
initsignal(psig);
P(IO);
  OpenOutput(‘DTA’);
  WriteString("start MultiP");
  WriteLn;
V(IO);
FOR i:=1 TO numP DO
  P(IO);
  WriteString("start dummy");
  WriteCard(i,3);
  WriteLn;
V(IO);
  start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
END;
FOR i:=1 TO numP DO
  P(IO);
  WriteString("wait psig");
  WriteCard(i,3);
  WriteLn;
V(IO);
  wait(psig)
END;
P(IO);
  WriteString("StatusOfScheduler = ");
  WriteInt(StatusOfScheduler(),3);
  WriteLn;
V(IO);
P(IO);
  WriteString("end MultiP");
  WriteLn;
  CloseOutput;
V(IO);
  HaltScheduler
END MultiProcessTest.
MODULE MultiProcessTest;

(* Written by Phil Rosine
8 Dec 83

Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADDRESS, ADR, SIZE, PROCESS, WORD;
FROM PreemptiveScheduler IMPORT wait, waited, signal, SIGNAL,
initsignal, start, HaltProcess, HaltScheduler,
P, V, initsemaphore, SEMAPHORE;
FROM InOut IMPORT WriteString, WriteLn, WriteCard,
OpenOutput, CloseOutput;

CONST
  numP = 10;

TYPE
  Stack = ARRAY [1..120] OF WORD;

VAR
  process: ARRAY [1..10] OF PROCESS;
  i : CARDINAL;
  workspace : ARRAY [1..10] OF Stack;
  psig : SIGNAL;
  curnum : CARDINAL;
  IO : SEMAPHORE;

PROCEDURE dummy;
  VAR Pnum : CARDINAL;
BEGIN
  INC(curnum); Pnum := curnum;
  P(IO);
  WriteString("dummy loop");WriteCard(Pnum,3);WriteLn;
  V(IO);
  P(IO);
  WriteString("dummy loop — signaling psig");WriteCard(Pnum,3);WriteLn;
  V(IO);
  IF waited(psig) THEN signal(psig);
  P(IO);
  WriteString("dummy loop — psig signaled");WriteCard(Pnum,3);WriteLn;
  V(IO);
  END;
  P(IO);
  WriteString("dummy loop — HaltProcess");WriteCard(Pnum,3);WriteLn;
  V(IO);
  HaltProcess
END dummy;

BEGIN (* MultiProcessTest *)
  initsemaphore(IO,1);
  curnum := 0;
P(IO);
OpenOutput("DTA");
WriteString("start MultiP");WriteLn;
V(IO);
initsignal(psig);
FOR i:=1 TO numP DO
  P(IO);
  WriteString("start dummy");WriteCard(i,3);WriteLn;
  V(IO);
  start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
END;
FOR i:=1 TO numP DO
  P(IO);
  WriteString("wait psig");WriteCard(i,3);WriteLn;
  V(IO);
  wait(psig)
END;

P(IO);
WriteString("halt scheduler");WriteLn;
V(IO);
HaltScheduler;
P(IO);
WriteString("end MultiP");WriteLn;
CloseOutput;
V(IO);
END MultiProcessTest.
MODULE MultiProcessTest;

(* Written by Phil Rosine
8 Dec 83

Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADDRESS, ADR, SIZE, PROCESS, WORD;
FROM PreemptiveScheduler IMPORT wait, waited, signal, SIGNAL,
initsignal, start, HaltProcess, HaltScheduler,
P, V, initsemaphore, SEMAPHORE;
FROM InOut IMPORT WriteString, WriteLn, WriteCard,
OpenOutput, CloseOutput;

CONST
numP = 10;
TYPE
Stack = ARRAY [1..100] OF WORD;
VAR
process: ARRAY [1..10] OF PROCESS;
i : CARDINAL;
workspace : ARRAY [1..10] OF Stack;
psig : SIGNAL;
curnum : CARDINAL;
IO : SEMAPHORE;
IOWork : ARRAY[1..4] OF WORD;

PROCEDURE dummy;
VAR Pnum : CARDINAL;
BEGIN
INC(curnum); Pnum := curnum;
P(IO);
WriteString("dummy loop");WriteCard(Pnum,3);WriteLn;
V(IO);
P(IO);
WriteString("dummy loop -- HaltProcess");WriteCard(Pnum,3);WriteLn;
V(IO);
HaltProcess
END dummy;

BEGIN (* MultiProcessTest *)
initsemaphore(IO,1,ADR(IOWork));
curnum := 0;
P(IO);
OpenOutput("DTA");
WriteString("start MultiP");WriteLn;
V(IO);
initsignal(psig);
FOR i:=1 TO numP DO
  P(IO);
WriteString("start dummy");WriteCard(i,3);WriteLn;
V(IO);
start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
END;
P(IO);
WriteString("wait psig");WriteLn;
V(IO);
wait(psig);
P(IO);
WriteString("end MultiP");WriteLn;
CloseOutput;
V(IO);
END MultiProcessTest.
MODULE MultiProcessTest;

(* Written by Phil Rosine
  8 Dec 83

  Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADDRESS, ADR, SIZE, PROCESS, WORD;
FROM PreemptiveScheduler IMPORT wait, waited, signal, SIGNAL,
initsignal, start, HaltProcess, HaltScheduler,
P, V, initsemaphore, SEMAPHORE;
FROM InOut IMPORT WriteString,WriteLn,WriteCard, OpenOutput, CloseOutput;

CONST
  numP = 10;

TYPE
  Stack = ARRAY [1..200] OF WORD;

VAR
  process: ARRAY [1..numP] OF PROCESS;
  i : CARDINAL;
  workspace : ARRAY [1..10] OF Stack;
  psig : SIGNAL;
  curnum : CARDINAL;
  IO : SEMAPHORE;
  IOwork : ARRAY [1..4] OF WORD;

PROCEDURE dummy;
VAR
  Pnum, j : CARDINAL;
BEGIN
  INC(curnum); Pnum := curnum;
P(IO);
  WriteString("dummy loop"); WriteCard(Pnum,3); WriteLn;
  V(IO);
  j := 0;
  LOOP
    INC(j);
    IF j>Pnum*100 THEN EXIT END
  END;
P(IO);
  WriteString("dummy loop -- HaltProcess"); WriteCard(Pnum,3); WriteLn;
  V(IO);
  HaltProcess
END dummy;

BEGIN (* MultiProcessTest *)
  initsemaphore(IO,1,ADR(IOwork));
  curnum := 0;
P(IO);
  OpenOutput("DTA");
WriteString("start MultiP");WriteLn;
V(IO);
initsignal(psig);
FOR i:=1 TO numP DO
  P(IO);
  WriteString("start dummy");WriteCard(i,3);WriteLn;
  V(IO);
  start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
END;
  P(IO);
  WriteString("wait psig");WriteLn;
  V(IO);
  wait(psig);
P(IO);
WriteString("end MultiP");WriteLn;
CloseOutput;
V(IO);
END MultiProcessTest.
MODULE MultiProcessTest;

(* Written by Phil Rosine  
8 Dec 83  
Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADDRESS, ADDR, SIZE, PROCESS, WORD;
FROM PreemptiveScheduler IMPORT wait, waited, signal, SIGNAL, 
initsignal, start, HaltProcess, HaltScheduler, 
P, V, initsemaphore, SEMAPHORE;
FROM InOut IMPORT WriteString, WriteLn, WriteCard, 
OpenOutput, CloseOutput;

CONST
  numP = 10;

TYPE
  Stack = ARRAY [1..100] OF WORD;

VAR
  process: ARRAY [1..10] OF PROCESS;
  i : CARDINAL;
  workspace : ARRAY [1..10] OF Stack;
  psig : SIGNAL;
  curnum : CARDINAL;
  I0 : SEMAPHORE;
  IOwork : ARRAY[1..4] OF WORD;

PROCEDURE dummy;
  VAR Pnum, j  : CARDINAL;
BEGIN
  INC(curnum); Pnum := curnum;
  P(IO);
  WriteString("dummy loop");WriteCard(Pnum,3);WriteLn;
  V(IO);
  LOOP  (* infinite loop *)
    j := 0;
    LOOP
      INC(j);
      IF j>Pnum*1000 THEN EXIT END
    END;
    P(IO);
    WriteString("dummy loop — signaling psig");WriteCard(Pnum,3);WriteLn;
    V(IO);
    IF waited(psig) THEN signal(psig);
    P(IO);
    WriteString("dummy loop — psig signaled");WriteCard(Pnum,3);WriteLn;
    V(IO);
  END;  (* infinite loop *)
END dummy;
BEGIN (* MultiProcessTest *)
    initsemaphore(IO,1,ADR(IOwork));
    curnum := 0;
    P(IO);
    OpenOutput("DTA");
    WriteString("start MultiP");WriteLn;
    V(IO);
    initsignal(psig);
    FOR i:=l TO numP DO
        P(IO);
        WriteString("start dummy");WriteCard(i,3);WriteLn;
        V(IO);
        start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
    END;
    FOR i:=l TO numP DO
        P(IO);
        WriteString("wait psig");WriteCard(i,3);WriteLn;
        V(IO);
        wait(psig)
    END;
    P(IO);
    WriteString("wait scheduler");WriteLn;
    V(IO);
    HaltScheduler;
    P(IO);
    WriteString("end MultiP");WriteLn;
    CloseOutput;
    V(IO);
END MultiProcessTest.
 MODULE MultiProcessTest;

 (* Written by Phil Rosine
  8 Dec 83

 Designed to test concurrent process interaction. *)

 FROM SYSTEM IMPORT ADDRESS, ADR, SIZE, PROCESS, WORD;
 FROM PreemptiveScheduler IMPORT wait, waited, signal, SIGNAL,
 initsignal, start, HaltProcess, HaltScheduler,
 P, V, initsemaphore, SEMAPHORE;
 FROM InOut IMPORT WriteString, WriteLn, WriteCard,
 OpenOutput, CloseOutput;

 CONST
 numP = 10;
 TYPE
 Stack = ARRAY [1..100] OF WORD;
 VAR
 process: ARRAY [1..10] OF PROCESS;
 i : CARDINAL;
 workspace : ARRAY [1..10] OF Stack;
 psig : SIGNAL;
 curnum : CARDINAL;
 IO : SEMAPHORE;
 IOwork : ARRAY[1..4] OF WORD;

 PROCEDURE dummy;
 VAR Pnum, j : CARDINAL;
 BEGIN
 INC(curnum); Pnum := curnum;
 P(IO);
 WriteString("dummy loop");WriteCard(Pnum,3);WriteLn;
 V(IO);
 LOOP (* infinite loop *)
 j := 0;
 LOOP
 INC(j);
 IF j>Pnum*100 THEN EXIT END
 END;
 P(IO);
 WriteString("dummy loop -- signaling psig");WriteCard(Pnum,3);WriteLn;
 V(IO);
 IF waited(psig) THEN signal(psig);
 P(IO);
 WriteString("dummy loop -- psig signaled");WriteCard(Pnum,3);WriteLn;
 V(IO);
 END (* infinite loop *)
 END dummy;
BEGIN (* MultiProcessTest *)
    initsemaphore(IO,1,ADR(IOwork));
    curnum := 0;
    P(IO);
    OpenOutput("DTA");
    WriteString("start MultiP");WriteLn;
    V(IO);
    initsignal(psig);
    FOR i:=1 TO numP DO
        P(IO);
        WriteString("start dummy");WriteCard(i,3);WriteLn;
        V(IO);
        start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
    END;
    FOR i:=1 TO numP DO
        P(IO);
        WriteString("wait psig");WriteCard(i,3);WriteLn;
        V(IO);
        wait(psig)
    END;
    P(IO);
    WriteString("halt scheduler");WriteLn;
    V(IO);
    HaltScheduler;
    P(IO);
    WriteString("end MultiP");WriteLn;
    CloseOutput;
    V(IO);
END MultiProcessTest.
MODULE MultiProcessTest;

(* Written by Phil Rosine
8 Dec 83

Designed to test concurrent process interaction. *)

FROM SYSTEM IMPORT ADDRESS, ADR, SIZE, PROCESS, WORD;
FROM PreemptiveScheduler IMPORT wait, waited, signal, SIGNAL, initsignal, start, HaltProcess, HaltScheduler, ProcessID, P, V, initsemaphore, SEMAPHORE, StatusOfScheduler;
FROM InOut IMPORT WriteString, WriteLn, WriteCard, OpenOutput, CloseOutput, WriteInt;

CONST	numP = 10;

TYPE
Stack = ARRAY [1..120] OF WORD;

VAR
process: ARRAY [1..10] OF PROCESS;
i : CARDINAL;
workspace : ARRAY [1..10] OF Stack;
psig : SIGNAL;
IO : SEMAPHORE;

PROCEDURE dummy;
BEGIN
P(IO);
WriteString("dummy loop");
WriteCard(ProcessID(),3);
WriteLn;
V(IO);
P(IO);
WriteString("dummy loop — waiting psig");
WriteCard(ProcessID(),3);
WriteLn;
V(IO);
wait(psig);
P(IO);
WriteString("dummy loop — HaltProcess");
WriteCard(ProcessID(),3);
WriteLn;
V(IO);
HaltProcess
END dummy;

BEGIN (* MultiProcessTest *)
initsemaphore(IO,1);
P(IO);
WriteString("start MultiP");
WriteLn;
V(IO);
initsignal(psig);
FOR i:=1 TO numP DO
  P(IO);
  WriteString("start dummy");
  WriteCard(i,3);
  WriteLn;
  V(IO);
  start(dummy,ADR(workspace[i]),SIZE(workspace[i]))
END;
FOR i:=1 TO numP DO
  P(IO);
  WriteString("wait psig");
  WriteCard(i,3);
  WriteLn;
  V(IO);
  wait(psig)
END;
P(IO);
WriteString("StatusOfScheduler = ");
WriteInt(StatusOfScheduler(),3);
WriteLn;
V(IO);
P(IO);
WriteString("end MultiP");
WriteLn;
V(IO);
END MultiProcessTest.
Monitor Pre-Processor Tests

The following programs were written to test various parts of the program MONITR as it was developed.

TString : : to test the string library
producerconsumer : : driver for the bounded buffer monitor
boundedbuffer : : a monitor; this includes the DEFINITION module, a MON file, a hand implemented version of the monitor, and the output from MONITR.

May 23, 1984

Philip E. Rosine
MODULE TString;

(* written by Phil Rosine
   22 Mar 84

   This module is used to test the strcmp and strncpy functions.

   Modified to test strncmp. 24 Mar 84 *)

FROM InOut IMPOR ReadString, WriteString, WriteLn;
FROM MLibrary IMPOR strncmp, strcmp, strncpy;

VAR
   strA, strB : ARRAY [0..80] OF CHAR;

BEGIN
   LOOP
      WriteString("Enter strA: ");
      ReadString(strA); WriteLn;
      IF strA[0] = 'Q' THEN EXIT END;
      WriteString("Enter strB: ");
      ReadString(strB); WriteLn;
      WriteLn;
      (* IF strcmp(strA,strB) THEN WriteString("TRUE")
         ELSE WriteString("FALSE") END; *)
      IF strncmp("test",strB,0,2,4)
         THEN WriteString("TRUE")
         ELSE WriteString("FALSE") END;
      IF strncmp(strA,strB,0,2,5)
         THEN WriteString("TRUE")
         ELSE WriteString("FALSE") END;
      WriteLn;
      (* strncpy(strA,strB,0,3,5);
         WriteString(strB);WriteLn *)
   END
END TString.
MODULE producerconsumer;

(* from Ben-Ari[1982] pp75-76
implemented in Modula-2 by Phil Rosine
Feb 84

Uses the Monitor boundedbuffer (BOUNDE.MON) which
is processed through the pre-processor MONITR. This
allows concurrency using the PreemptiveScheduler
module. This is a test program for the Monitor
Pre-Processor.
*)

FROM SYSTEM IMPORT WORD,ADR,SIZE;
FROM PreemptiveScheduler IMPORT start, wait, SIGNAL,
TraceOutput, initsignal;
FROM InOut IMPORT Writeln, WriteInt, WriteString;
FROM boundedbuffer IMPORT append, take;

VAR
  awhile: SIGNAL;
  (* added to allow the main program to pause *)
  Pwsp, Cwsp: ARRAY [0..120] OF WORD;

PROCEDURE producer;
VAR
  v: INTEGER;
BEGIN
  v := 0;
  REPEAT
    INC(v); (* produce(v) *)
    append(v)
  UNTIL v>10
  (**ASSERT: when all items have been appended to the
  buffer, producer will reach this point and it process
  will end normally. **) 
END producer;

PROCEDURE consumer;
VAR
  v: INTEGER;
BEGIN
  (**ASSERT: this process will be an infinite loop. **) 
  LOOP 
    take(v);
    WriteInt(v,5); WriteLn
  END
  (**ASSERT: Since this is an infinite process, this point
  (normal end) will never be reached. Process consumer will
  end only when it has taken all the items from the buffer
  *)
and producer has ended. Since the main program will be in a wait state (wait(awhile)), at that time the scheduler will force a halt and a return to the main since there will be no process ready to run. **

END consumer;

BEGIN (* main program *)
initsignal(awhile);
(* cobegin *)
start(producer,ADR(Pwsp),SIZE(Pwsp));
start(consumer,ADR(Cwsp),SIZE(Cwsp));
(* coend *)

(**ASSERT: At this point both of the sup-processes have been started and are running. This process (the main) must wait until producer and consumer both finish. To do this, it is forced to wait on a SIGNAL. **)
wait(awhile);

(**ASSERT: The SIGNAL awhile will never be formally released (signaled). This point in the program will be reached when producer has ended normally, and when consumer has taken everything from the buffer. At that time boundedbuffer.take will cause consumer to wait, and since there will then be no processes ready to run, the scheduler will force a halt and return to this point in the main. **)
TraceOutput(15, "DK PRODUCTRC")

END producerconsumer.
DEFINITION MODULE boundedbuffer;

(* from Ben-Ari[1982] pp75-76
implemented in Modula-2 by Phil Rosine
Feb 84

This is the monitor from program 'producerconsumer'.
It will be used as a test case for the Monitor Pre-
Processor of the Modula system. Initially it will
be hand programmed both as a monitor (using condition
typed variables) and as a module using semaphores.
When the pre-processor is ready, it will be used
to test its action.
*)

EXPORT QUALIFIED append, take;

PROCEDURE appendCv:INTEGER);
(* if the buffer is not full, add v to it *)

PROCEDURE take(VAR v:INTEGER);
(* if the buffer is not empty, remove v from it *)

END boundedbuffer.
IMPLEMENTATION MODULE boundedbuffer;

   (* from Ben-Ari[1982] pp75-76
      implemented in Modula-2 by Phil Rosine
      Feb 84

   This is the monitor from program 'producerconsumer'.
   It will be used as a test case for the Monitor Pre-
   Processor of the Modula system. Initially it will
   be hand programmed both as a monitor (using CONDITION
   typed variables) and as a module using semaphores.
   When the pre-processor is ready, it will be used
   to test its action. *)

CONST
   sizeofbuffer = 4;
   (* Ben-Ari has this global to the program. Since it
      is logically a part of the monitor rather than the
      program proper, it is declared here rather than in
      the program. *)

VAR
   b: ARRAY [0..sizeofbuffer] OF INTEGER;
      (* circular buffer *)
   in: INTEGER;
      (* pointer to b for adding items to the buffer *)
   out: INTEGER;
      (* pointer to b for removing items *)
   n: INTEGER;
      (* count of items in the buffer *)
   notempty, notfull: CONDITION;

PROCEDURE append(v:INTEGER);
   (* if the buffer is not full, add v to it *)
BEGIN
   IF n=(sizeofbuffer+1) THEN (* the buffer is full *)
      WAIT(notfull)
   END;
   b[in] := v;
   in := in + 1;
   IF in=(sizeofbuffer+1) THEN (* circular buffer *)
      in := 0;
   END;
   n := n + 1;
   SIGNAL(notempty)
END append;

PROCEDURE take(VAR v:INTEGER);
(* if the buffer is not empty, remove v from it *)
BEGIN
  IF n=0 THEN (* the buffer is empty *)
    WAIT(notempty)
  END;
  v := b[out];
  out := out + 1;
  IF out=(sizeofbuffer+1) THEN (* circular buffer *)
    out := 0
  END;
  n := n - 1;
  SIGNAL(notfull)
END take;

BEGIN (* monitor body *)
  in := 0;
  out := 0;
  n := 0
END boundedbuffer.
IMPLEMENTATION MODULE boundedbuffer;
FROM PreemptiveScheduler IMPORT SEMAPHORE, P, V, initsemaphore,
SIGNAL, signal, wait, waited, initsignal;

(* from Ben-Ari[1982] pp75-76
implemented in Modula-2 by Phil Rosine
Feb 84

This is the monitor from program 'producerconsumer'.
It will be used as a test case for the Monitor Pre-
Processor of the Modula system. Initially it will
be hand programmed both as a monitor (using condition
typed variables) and as a module using semaphores.
When the pre-processor is ready, it will be used
to test its action.
*)

(*
This is a hand-built version of the monitor with the
'release' type and operations replaced by SIGNAL
and SEMAPHORE types and operations. This is to test
the results of pre-processor program MONITR before
it is actually written.
*)

CONST
  sizeofbuffer = 10;
  (* Ben-Ari has this global to the program. Since it
  is logically a part of the monitor rather than the
  program proper, it is declared here rather than in
  the program. *)

VAR
  MONITORSEM: SEMAPHORE;
  b: ARRAY [0..sizeofbuffer] OF INTEGER;
  (* circular buffer *)
  in: INTEGER;
  (* pointer to b for adding items to the buffer *)
  out: INTEGER;
  (* pointer to b for removing items *)
  n: INTEGER;
  (* count of items in the buffer *)
  notempty, notfull: SIGNAL (*condition*);

PROCEDURE append(v:INTEGER);
  (* if the buffer is not full, add v to it *)
BEGIN
  P(MONITORSEM);
  IF n=(sizeofbuffer+1) THEN (* the buffer is full *)
    V(MONITORSEM); wait(notfull)
END;
  b[in] := v;
  in := in + 1;
  IF in=(sizeofbuffer+1) THEN (* circular buffer *)
    in := 0;
  END;
  n := n + 1;
  IF waited(notempty) THEN signal(notempty) ELSE V(MONITORSEM) END
END append;

PROCEDURE take(VAR v:INTEGER);
  (* if the buffer is not empty, remove v from it *)
BEGIN
  P(MONITORSEM);
  IF n=0 THEN (* the buffer is empty *)
    V(MONITORSEM);wait(notempty)
  END;
  v := b[out];
  out := out + 1;
  IF out=(sizeofbuffer+1) THEN (* circular buffer *)
    out := 0
  END;
  n := n - 1;
  IF waited(notfull) THEN signal(notfull) ELSE V(MONITORSEM) END
END take;

BEGIN (* monitor body *)
initsemaphore(MONITORSEM,1);
initsignal(notempty);
initsignal(notfull);
in := 0;
out := 0;
FOR n:=0 TO sizeofbuffer DO b[n]:=0 END;
n := 0
END boundedbuffer.
IMPLEMENTATION MODULE boundedbuffer;

(* from Ben-Ari[1982] pp75-76
implemented in Modula-2 by Phil Rosine
Feb 84

This is the monitor from program 'producerconsumer'.
It will be used as a test case for the Monitor Pre-
Processor of the Modula system. Initially it will
be hand programmed both as a monitor (using CONDITION
typed variables) and as a module using semaphores.
When the pre-processor is ready, it will be used
to test its action.
*)

(*** This module has been processed by the monitor pre-processor
MONITR. A number of lines have been added to the code by the
pre-processor. These lines are preceded and followed by comment
lines with three asterisks like this comment. ***)

(*** import ***)
FROM PreemptiveScheduler IMPORT SEMAPHORE, P, V, initsemaphore;
(*** end import ***)

CONST
  sizeofbuffer = 4;
  (* Ben-Ari has this global to the program. Since it
   is logically a part of the monitor rather than the
   program proper, it is declared here rather than in
   the program. *)

TYPE

  (*** type CONDITION ***)
  CONDITION = RECORD
    c :  SEMAPHORE;
    cc :  CARDINAL
    END;
  (*** end type CONDITION ***)

VAR
  (*** mutex variable declarations ***)
  MONITORSEM :  SEMAPHORE;
  MONITORCOND :  CONDITION;
  (*** end mutex variable declarations ***)

  b: ARRAY [0..sizeofbuffer] OF INTEGER;
    (* circular buffer *)
  in: INTEGER;
    (* pointer to b for adding items to the buffer *)
  out: INTEGER;
    (* pointer to b for removing items *)
n: INTEGER;
    (* count of items in the buffer *)
notempty, notfull: CONDITION;

(*** cond procedure declarations ***)
PROCEDURE WAIT(VAR C : CONDITION);
BEGIN
    INC(C.cc);
    IF MONITORCOND.cc > 0
        THEN V(MONITORCOND.c)
        ELSE V(MONITOREM)
    END;
    P(C.c);
    DEC(C.cc)
END WAIT;
(*** ***)
PROCEDURE SIGNAL(VAR C : CONDITION);
BEGIN
    INC(MONITORCOND.cc);
    IF C.cc > 0
        THEN
            V(C.c);
            P(MONITORCOND.c)
        END;
    DEC(MONITORCOND.cc)
END SIGNAL;
(*** ***)
PROCEDURE initcondition(VAR C : CONDITION);
BEGIN
    initsemaphore(C.c,0);
    C.cc := 0
END initcondition;
(*** end cond procedure declarations ***)

PROCEDURE append(v:INTEGER);
    (* if the buffer is not full, add v to it *)
BEGIN
    (** entry ***)
    P(MONITOREM);
    (** end entry ***)
    IF n=(sizeofbuffer+1) THEN (* the buffer is full *)
        WAIT(notfull)
    END;
    b[in] := v;
    in := in + 1;
    IF in=(sizeofbuffer+1) THEN (* circular buffer *)
        in := 0;
    END;
BEGIN

(* monitor body *).

in := 0;
out := 0;
in := 0
END boundedbuffer.
Appendix C

Scheduler Trace Tests

The following programs were written to test the trace feature for module PreemptiveScheduler. The program producerconsumer (used to test the MONITR program) was used to test the trace.

produc.map :: link map for producerconsumer
produc.trc :: trace output file
Storage map of layer producerconsumer (module from file DK:PRODUC.LNK) linked to layer ResidentMonitor from file SY:MODULA.M2S

separate module FilePool from file DK:FilePo.LNK
Data: 014270B
Proc # 0: 014272B Proc # 1: 014316B

separate module FileNames from file SY:FileNa.LNK
Data: 014452B
Proc # 0: Proc # 1: 014472B Proc # 2: 015324B
Proc # 3: 015622B Proc # 4: 016002B Proc # 5: 016330B

separate module Storage from file SY:Storag.LNK
Data: 016524B
Proc # 0: 026540B Proc # 1: 026574B Proc # 2: 027272B
Proc # 3: 027374B

separate module Streams from file SY:Stream.LNK
Data: 027524B
Proc # 0: Proc # 1: 027524B Proc # 2: 027630B
Proc # 3: 027702B Proc # 4: 027734B Proc # 5: 030030B
Proc # 6: 030114B Proc # 7: 030222B Proc # 8: 030310B
Proc # 12: 031036B Proc # 13: 031162B

separate module FInOut from file DK:FInOut.LNK
Data: 031452B
Proc # 0: 031670B Proc # 1: 031776B Proc # 2: 032202B
Proc # 3: 032624B Proc # 4: 032732B Proc # 5: 033060B
Proc # 6: 033202B Proc # 7: 033412B Proc # 8: 034262B
Proc # 9: 035024B Proc # 10: 035142B Proc # 11: 035210B
Proc # 15: 036174B Proc # 16: 036432B Proc # 17: 036676B
Proc # 18: 037742B Proc # 19: 040070B Proc # 20: 040310B
Proc # 21: 040442B Proc # 22: 041474B

separate module PreemptiveScheduler from file DK:TRACES.LNK
Data: 042230B
Proc # 0: 050512B Proc # 1: 050726B Proc # 2: 050766B
Proc # 3: 051032B Proc # 4: 051360B Proc # 5: 051452B
Proc # 6: 051634B Proc # 7: 052230B Proc # 8: 052266B
Proc # 9: 052510B Proc # 10: 053012B Proc # 11: 053500B
Proc # 15: 054760B Proc # 16: 055334B

separate module InOut from file DK:InOut.LNK
Data: 055450B
Proc # 0: 056142B Proc # 1: 056200B Proc # 2: 056754B
Proc # 3: 060104B Proc # 4: 060162B Proc # 5: 060252B
Proc # 6: 060450B Proc # 7: 060736B Proc # 8: 061644B
Proc # 15: 063452B Proc # 16: 063664B Proc # 17: 064122B
Proc # 18: 064200B Proc # 19: 064374B

separate module boundedbuffer from file DK:bounde.LNK
Data: 064516B
Proc # 0: 064600B  Proc # 1: 064706B  Proc # 2: 065106B
Proc # 3: 065304B  Proc # 4: 065410B  Proc # 5: 065472B
separate module producerconsumer from file DK:PRODUC.LNK
Data: 065532B
Proc # 0: 066516B  Proc # 1: 066652B  Proc # 2: 066720B
First free location: 066772B
Appendix D

Level 0 PSA Formatted Problem Statement
Name Selection

Parameters: DB=MODPR3.DBF PRINT PUNCH=PSANAM.TMP EMPTY
SELECTION="TRACE-KEY="Level-0" ORDER=BYTYPE

<table>
<thead>
<tr>
<th>1</th>
<th>Development-Test-Data</th>
<th>ENTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Test-Run-Results</td>
<td>ENTITY</td>
</tr>
<tr>
<td>3</td>
<td>Requests</td>
<td>INPUT</td>
</tr>
<tr>
<td>4</td>
<td>User-Test-Data</td>
<td>INPUT</td>
</tr>
<tr>
<td>5</td>
<td>User</td>
<td>INTERFACE</td>
</tr>
<tr>
<td>6</td>
<td>Final-Program</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>7</td>
<td>Compiler-Linker</td>
<td>PROCESS</td>
</tr>
<tr>
<td>8</td>
<td>Pre-Processing</td>
<td>PROCESS</td>
</tr>
<tr>
<td>9</td>
<td>Program-Development</td>
<td>PROCESS</td>
</tr>
<tr>
<td>10</td>
<td>Run-Time-System</td>
<td>PROCESS</td>
</tr>
<tr>
<td>11</td>
<td>Libraries</td>
<td>SET</td>
</tr>
<tr>
<td>12</td>
<td>Object-Code</td>
<td>SET</td>
</tr>
<tr>
<td>13</td>
<td>Source-Code</td>
<td>SET</td>
</tr>
<tr>
<td>14</td>
<td>Special-Files</td>
<td>SET</td>
</tr>
</tbody>
</table>
DEFINE ENTITY Development-Test-Data;
DESCRIPTION;
This is extra test data generated by the Program-Develo... process. It is based upon data collected during analysis and tailored to suit the implementation.

TRACE-KEY IS: 'Level-0';
ADDED TO:
  Test-Data BY Program-Development;
DERIVED BY: Program-Development
USING: Requests,
       User-Test-Data;
DEFINE ENTITY Test-Run-Results;
DESCRIPTION;
Results are the output of a program being tested. This may include error messages, etc.
;
TRACE-KEY IS: 'Level-0';
DERIVED BY: Run-Time-System
USING: Test-Data,
Object-Code;
USED BY:
   Program-Development
   TO DERIVE Final-Program;
USED BY:
   Program-Development
   TO UPDATE Source-Code;
USED BY:
   Program-Development
   TO UPDATE Special-Files;
Formatted Problem Statement

DEFINE INPUT Requests;

DESCRIPTION;
A Request consists of any number of things, from vague
ideas and suggestions by a user to formal requests for
programs.

TRACE-KEY IS: ‘Level-0’;
GENERATED:   BY User;
RECEIVED:    BY Program-Development;
USED BY:
  Program-Development
    TO DERIVE Development-Test-Data;
USED BY:
  Program-Development
    TO DERIVE Final-Program;
USED BY:
  Program-Development
    TO DERIVE Source-Code;
USED BY:
  Program-Development
    TO DERIVE Special-Files;
DEFINE INPUT User-Test-Data;
DESCRIPTION;
The User-Test-Data is data given by the User to the programmer/analyst who is writing the program. It may require more processing during program development in order to serve as Test-Data for the program.

TRACE-KEY IS: 'Level-0';
GENERATED: BY User;
RECEIVED: BY Program-Development;
USED BY:
Program-Development
TO DERIVE Development-Test-Data;
DEFINE INTERFACE User;
SYNONYMS ARE: Instructor;
DESCRIPTION;
A User is the individual or group in the real world which requests program development or problem solving to be done by the Program Development process. In the context of a school, the User would be the Instructor. Interaction between the User and the Development process can be very intricate — only a minimal interaction is shown by this problem statement.

TRACE-KEY IS: `Level-0`;
GENERATES:
Requests,
User-Test-Data;
RECEIVES:
Final-Program;
UNIVERSITY OF MONTANA PSA/PSL

Formatted Problem Statement

DEFINE OUTPUT Final-Program;
DESCRIPTION;
The Final-Program has been tested and approved by both the development personnel and the User.

TRACE-KEY IS: 'Level-0';
GENERATED: BY Program-Development;
RECEIVED: BY User;
DERIVED BY: Program-Development
USING: Test-Run-Results,
Requests,
Test-Data;
DEFINE PROCESS Compiler-Linker;

DESCRIPTION;
The compiler and linker are employed during Program-
Development to process the Source-Code into Object-
Code which can be executed by the Run-Time-System.

KEYWORDS ARE: 'Phase.1',
               'process.2';
TRACE-KEY IS: 'Level-0';
SUBPARTS ARE: Compiler,
              Linker;
DERIVES: Object-Code
USING: Source-Code;
EMPLOYS: Libraries;
UTILIZED BY: Program-Development;
DEFINE PROCESS Pre-Processing;
DESCRIPTION;
This process takes source-code containing non-Modula-2 statements (perhaps Ada rendezvous constructs or guarded statements) and converts the code into a Pre-Processed-Source which contains the correct calls to subroutines implementing the desired functions.

KEYWORDS ARE: 'Phase.II',
               'process.1';
TRACE-KEY IS: 'Level-0';
SUBPARTS ARE: Monitor-Processor;
DERIVES:     Source-Code
USING:       Special-Files;
EMPLOYS:     Libraries;
Formatted Problem Statement

DEFINE PROCESS Program-Development;
DESCRIPTION;
The Program-Development process is the sequence which many authors call the System-Life-Cycle. The Modula-2 project will really only be concerned with the processes and data sets used during the implementation and testing phases of this cycle.

TRACE-KEY IS: 'Level-0';
GENERATES:
   Final-Program;
RECEIVES:
   Requests,
   User-Test-Data;
SUBPARTS ARE: Analysis,
   Design,
   Implementation/Test,
   Verification/Validation;
ADDS:
   Development-Test-Data TO Test-Data;
DERIVES:
   Development-Test-Data
USING:
   Requests,
   User-Test-Data;
DERIVES:
   Final-Program
USING:
   Test-Run-Results,
   Requests,
   Test-Data;
DERIVES:
   Source-Code
USING:
   Requests;
DERIVES:
   Special-Files
USING:
   Requests;
UPDATES:
   Source-Code
USING:
   Test-Run-Results;
UPDATES:
   Special-Files
USING:
   Test-Run-Results;
EMPLOYS:
   Libraries;
UTILIZES:
   Compiler-Linker,
   Run-Time-System;
 unfairly unfair
DEFINE SET Libraries;
DESCRIPTION;
The Libraries are the set of 'standard' subroutines which are called in by the Compiler and Linker with or without specific requests by the Programmer.
;  TRACE-KEY IS: 'Level-0';
SUBSETS ARE: IOControl,
InOut,
PreemptiveScheduler,
RealInOut;
EMPLOYED BY: Compiler-Linker,
Pre-Processing,
Program-Development;
DEFINE SET Object-Code;
TRACE-KEY IS: 'Level-0';
DERIVED BY: Compiler-Linker
USING: Source-Code;
DERIVED BY: Linker
USING: Symbol-File,
        Linker-File;
USED BY:
        Run-Time-System
        TO DERIVE Test-Run-Results;
Formatted Problem Statement

DEFINE SET Source-Code;
DESCRIPTION;
Source-Code files are the Modula-2 statements which are suitable for compilation.
;
TRACE-KEY IS: 'Level-0',
'Level-1-1',
'Level-1-2';

SUBSETS ARE: Definition-Module-Code,
Implementation-Module-Code,
Program-Module-Code;

DERIVED BY: Program-Development
USING: Requests;

DERIVED BY: Pre-Processing
USING: Special-Files;

USED BY:
    Compiler
    TO DERIVE List-File;

USED BY:
    Compiler-Linker
    TO DERIVE Object-Code;

UPDATED BY: Program-Development
USING: Test-Run-Results;
DEFINE SET       Special-Files;
   DESCRIPTION;
Special-Files may contain non-Modula-2 source statements
which require conversion into Modula statements or sub-
routine calls. After this is done, pre-processed
code is produced which can be run through the standard
compiler.
;
   TRACE-KEY IS: 'Level-0';
   SUBSETS ARE: Monitor-Code;
   DERIVED BY: Program-Development
   USING: Requests;
   USED BY: Pre-Processing
   TO DERIVE Source-Code;
   UPDATED BY: Program-Development
   USING: Test-Run-Results;
Index

1 Analysis 10
2 Compiler 8, 14
3 Compiler-Linker 8, 10, 12, 13, 14
4 Definition-Module-Code 14
5 Design 10
6 Development-Test-Data 2, 4, 5, 10(2)
7 Final-Program 3, 4, 6, 7, 10(2)
8 IOControl 12
9 Implementation-Module-Code 14
10 Implementation/Test 10
11 InOut 12
12 Instructor 6
13 Libraries 8, 9, 10, 12
14 Linker 8, 13
15 Linker-File 13
16 List-File 14
17 Monitor-Code 15
18 Monitor-Processor 9
19 Object-Code 3, 8, 11, 13, 14
20 Pre-Processing 9, 12, 14, 15
21 PreemptiveScheduler 12
22 Program-Development 2(2), 3(3), 4(5), 5(2), 7(2), 8, 10, 11, 12, 14(2), 15(2)
23 Program-Module-Code 14
## Index

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Description</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>RealInOut</td>
<td>12</td>
</tr>
<tr>
<td>25</td>
<td>Requests</td>
<td>2, 4, 6, 7, 10(5), 14, 15</td>
</tr>
<tr>
<td>26</td>
<td>Run-Time-System</td>
<td>3, 10, 11, 13</td>
</tr>
<tr>
<td>27</td>
<td>Source-Code</td>
<td>3, 4, 8, 9, 10(2), 13, 14, 15</td>
</tr>
<tr>
<td>28</td>
<td>Special-Files</td>
<td>3, 4, 9, 10(…), 14, 15</td>
</tr>
<tr>
<td>29</td>
<td>Symbol-File</td>
<td>13</td>
</tr>
<tr>
<td>30</td>
<td>Test-Data</td>
<td>2, 3, 7, 10(…), 11</td>
</tr>
<tr>
<td>31</td>
<td>Test-Run-Results</td>
<td>3, 7, 10(3), 11, 13, 14, 15</td>
</tr>
<tr>
<td>32</td>
<td>User</td>
<td>4, 5, 6, 7</td>
</tr>
<tr>
<td>33</td>
<td>User-Test-Data</td>
<td>2, 5, 6, 10(2)</td>
</tr>
<tr>
<td>34</td>
<td>Verification/Validation</td>
<td>10</td>
</tr>
</tbody>
</table>
This report presents, in graphical form, the data entering or leaving a PROCESS or INTERFACE. The PROCESS or INTERFACE in question is represented as a box centered in the page. A SYNONYM and a value for a user-specified ATTRIBUTE ("NODE-ATTRIBUTE") may appear in the box below the basic name of the PROCESS or INTERFACE.

Data objects entering (displayed on the left or above the box, depending upon page width) are determined from the RECEIVES, EMPLOYS, USED TO DERIVE, USED TO UPDATE, and optionally UPDATES relationships.

Data objects leaving (displayed on the right or below the box, depending upon page width) are determined from the DERIVES, GENERATES, and UPDATES relationships.

ATTRIBUTE values of the PROCESS or INTERFACE may also be displayed as the top or bottom sections of the box.

The height of the box is set by the greater of the numbers of data objects entering and leaving the PROCESS or INTERFACE.
Function Flow Data Diagram

- Final-Program

+--------------------------------------+
| User | Instructor | I |
+--------------------------------------+

- Requests
- User-Test-Data
Function Flow Data Diagram

- Libraries
- Source-Code

+-----------------------------+
I Compiler-Linker I
I
+-----------------------------+

- Object-Code
Function Flow Data Diagram

- Libraries
- Special-Files

+--------------------------------------+
| Pre-Processing |
+--------------------------------------+
Function Flow Data Diagram

- Requests
- User-Test-Data
- Libraries
- Test-Run-Results
- Test-Data

I

I

I

I

I

I

I

Program-Development

I

I

I

I

I

I

I

Final-Program

Development-Test-Data

Source-Code

Special-Files
Function Flow Data Diagram

- Test-Data
- Object-Code

+--------------------------------------+
I        Run-Time-System        I
I
+--------------------------------------+

- Test-Run-Results
Appendix D

Level-1 PSA Formatted Problem Statement
Name Selection

Parameters: DB=MODPR3.DBF PRINT PUNCH=PSANAMTMP EMPTY

SELECTION= "TRACE-KEY="Level-1" ORDER=BYTYPE

1  Compiler   PROCESS
2  Linker     PROCESS
3  Monitor-Processor  PROCESS
4  Definition-Module-Code  SET
5  Implementation-Module-Code  SET
6  Linker-File  SET
7  List-File  SET
8  Program-Module-Code  SET
9  Reference-File  SET
10 Symbol-File  SET
Parameters:  DB=MODPR3.DBF FILE=PSANAM.TMP INDEX
NOPUNCED-NAMES PRINT NOPUNCH  SMARG=5 NMARG=20
AMARG=10 BMARG=25 HMARG=57 CMARG=1  HMARG=18
ONE-PER-LINE COMMENT NEW-PAGE NONEW-LINE
NOALL-STATEMENTS COMPLEMENTARY-STATEMENTS
NOLINE-NUMBERS NOPRINTEOF NODLC-COMMENT SORT-NAME-LIST

DEFINE PROCESS Compiler;

DESCRIPTION;
The Compiler analyzes the source-code modules for errors
and produces an error listing or linker/symbol-Files. It
produces relocatable code information. A List-File is
produced even if errors do not occur.

KEYWORDS ARE: 'Phase.I',
'process.2.1';

TRACE-KEY IS: 'Level-1',
'Level-1-2';

SUBPARTS ARE: Compiler-Base,
Compiler-Initialization,
Compiler-Lister,
Pass-1,
Pass-2,
Pass-3,
Pass-4,
Pass-5,
Symbol-File-Generator;

PART OF: Compiler-Linker;

DERIVES: Symbol-File

USING: Definition-Module-Code;

DERIVES: List-File
USING: Source-Code;

DERIVES: Reference-File
USING: Program-Module-Code,
Implementation-Module-Code;

DERIVES: Linker-File
USING: Program-Module-Code,
Implementation-Module-Code;

EMPLOYs: ASCII-Identifier-Table,
Interpass-File-1,
Interpass-File-2;
DEFINE PROCESS  Linker;
  DESCRIPTION;
The Linker links several separate modules together to
build an overlay layer (program). It produces absolute
(position dependent) code (core images or Object-Code).

; KEYWORDS ARE: 'Phase.I',
                   'process.2.2';
TRACE-KEY IS:   'Level-1',
               'Level-1-2';
PART OF:       Compiler-Linker;
DERIVES:       Object-Code
USING:         Symbol-File,
               Linker-File;
DEFINE PROCESS Monitor-Processor;
SYNONYMS ARE: MONITR;
DESCRIPTION;
The Monitor-Processor takes a Special-File containing
Monitor-Code and transforms it into an Implementation-
Module which can be processed by the Compiler. This
involves adding an IMPORT statement, adding mutual-
exclusion controls to the exported procedures, and
converting conditions and operations on them into
SEMAPHORES and SEMAPHORE operations.

KEYWORDS ARE: 'Phase.II',
'process.1.1';
TRACE-KEY IS: 'Level-1',
'Level-1-1';
SUBPARTS ARE: Condition-Processing,
DEFINITION-processing,
Monitor-Setup;
PART OF: Pre-Processing;
DERIVES: Implementation-Module-Code
USING: Monitor-Code;
EMPLOYS: Definition-Module-Code;
DEFINE SET Definition-Module-Code;
SYNONYMS ARE: XXX.DEF;
DESCRIPTION;
The DEFINITION MODULE in Modula-2 defines all EXPORTED data constants, types, and variables and procedures. When another module IMPORTS something, the XXX.DEF file for that module is checked to insure that the item is EXPORTED and the definition is used for type checking.

TRACE-KEY IS: 'Level-1',
'Level-1-1',
'Level-1-2',
'Level-2-1.1';

SUBSET OF: Source-Code;

USED BY:
Compiler
TO DERIVE Symbol-File;

USED BY:
DEFINITION-processing
TO DERIVE exported-procedures-list;

EMPLOYED BY: Monitor-Processor;
DEFINe SET Implementation-Module-Code;
SYNONYMS ARE: XXX.MOD;

DESCRIPTION;
This module gives the implementation for the procedures
which are exported by the modules (as defined by XXX.D...
The XXX.MOD is compiled separately, and the linker res...
the connections to modules which import from it.

TRACE-KEY IS: 'Level-1',
               'Level-1-1',
               'Level-1-2',
               'Level-2-1.1';

SUBSET OF: Source-Code;
DERIVED BY: Condition-Processing
USING: Monitor-Code;
DERIVED BY: Monitor-Processor
USING: Monitor-Code;
USED BY:
Compiler
    TO DERIVE Reference-File;
USED BY:
Compiler
    TO DERIVE Linker-File;
Formatted Problem Statement

DEFINE SET Linker-File;
TRACE-KEY IS: 'Level-1', 'Level-1-2';
DERIVED BY: Compiler
USING: Program-Module-Code, Implementation-Module-Code;
USED BY: Linker
TO DERIVE Object-Code;
Formatted Problem Statement

DEFINE SET List-File;
TRACE-KEY IS: 'Level-1',
         'Level-1-2';
DERIVED BY: Compiler
USING: Source-Code;
DEFINE SET Program-Module-Code;
DESCRIPTION;
Program MODULEs are the top level of the Modula-2 hierarchy. The Source-Code for these will have a .MOD extension (like Implementation-Module-Code).

TRACE-KEY IS: 'Level-1',
              'Level-1-2';
SUBSET OF: Source-Code;
USED BY:
  Compiler
  TO DERIVE Reference-File;
USED BY:
  Compiler
  TO DERIVE Linker-File;
UNIVERSITY OF MONTANA PSA/PSL

Formatted Problem Statement

DEFINE SET Reference-File;
TRACE-KEY IS: 'Level-1',
'Level-1-2';
DERIVED BY: Compiler
USING: Program-Module-Code,
Implementation-Module-Code;
DEFINE SET Symbol-File;
TRACE-KEY IS: "Level-1",
"Level-1-2";
DERIVED BY: Compiler
USING: Definition-Module-Code;
USED BY:
   Linker
   TO DERIVE Object-Code;
Index

1 ASCII-Identifier-Table 2

2 Compiler 2, 5, 6(2), 7, 8, 9(2), 10, 11

3 Compiler-Base 2

4 Compiler-Initialization 2

5 Compiler-Linker 2, 3

6 Compiler-Lister 2

7 Condition-Processing 4, 6

8 DEFINITION-processing 4, 5

9 Definition-Module-Code 2, 4, 5, 11

10 Implementation-Module-Code 2(2), 4, 6, 7, 10

11 Interpass-File-1 2

12 Interpass-File-2 2

13 Linker 3, 7, 11

14 Linker-File 2, 3, 6, 7, 9

15 List-File 2, 8

16 MONITR 4

17 Monitor-Code 4, 6(2)

18 Monitor-Processor 4, 5, 6

19 Monitor-Setup 4

20 Object-Code 3, 7, 11

21 Pass-1 2

22 Pass-2 2

23 Pass-3 2

24 Pass-4 2

25 Pass-5 2
## Index

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Pre-Processing</td>
</tr>
<tr>
<td>27</td>
<td>Program-Module-Code</td>
</tr>
<tr>
<td>28</td>
<td>Reference-File</td>
</tr>
<tr>
<td>29</td>
<td>Source-Code</td>
</tr>
<tr>
<td>30</td>
<td>Symbol-File</td>
</tr>
<tr>
<td>31</td>
<td>Symbol-File-Generator</td>
</tr>
<tr>
<td>32</td>
<td>XXX.DEF</td>
</tr>
<tr>
<td>33</td>
<td>XXX.MOD</td>
</tr>
<tr>
<td>34</td>
<td>exported-procedures-list</td>
</tr>
</tbody>
</table>
Function Flow Data Diagram

Parameters:  DB=MODPR3.DBF  FILE=PSANAM.TMP  NOTOP-ATTRIBUTE  NOBOTTOM-ATTRIBUTE  NONODE-ATTRIBUTE  MISSING-ATTRIBUTE-VALUE='N/A'  BOX-CHARACTER='I-++++'  SYNONYM  NOUPDATED-AS-INPUT  NODATA-OUT-DASHES  NOINDEX  NOPUNCHED-NAMES  NEW-PAGE  EXPLANATION  DATA-IN-ORDER=STANDARD  DATA-OUT-ORDER=STANDARD  NOPLOT

This report presents, in graphical form, the data entering or leaving a PROCESS or INTERFACE. The PROCESS or INTERFACE in question is represented as a box centered in the page... SYNONYM and a value for a user-specified ATTRIBUTE ("NODE-ATTRIBUTE") may appear in the box below the basic name of the PROCESS or INTERFACE.

Data objects entering (displayed on the left or above the box, depending upon page width) are determined from the RECEIVES, EMPLOYS, USED TO DERIVE, USED TO UPDATE, and optionally UPDATES relationships.

Data objects leaving (displayed on the right or below the box, depending upon page width) are determined from the DERIVES, GENERATES, and UPDATES relationships.

ATTRIBUTE values of the PROCESS or INTERFACE may also be displayed as the top or bottom sections of the box.

The height of the box is set by the greater of the numbers of data objects entering and leaving the PROCESS or INTERFACE.
Function Flow Data Diagram

- Interpass-File-1
- Interpass-File-2
- ASCII-Identifier-Table
- Definition-Module-Code
- Source-Code
- Program-Module-Code
- Implementation-Module-Code

+------------------------------------+
| I                                |
| I                                |
| I                                |
| I                                |
| I                                |
| I                                |

Compiler

| I |
| I |
| I |
| I |

+------------------------------------+

- Symbol-File
- List-File
- Reference-File
- Linker-File
Function Flow Data Diagram

- Symbol-File
- Linker-File

+-----------------------------+
I                  Linker      I
I
+-----------------------------+

- Object-Code
Function Flow Data Diagram

- Definition-Module-Code
- Monitor-Code

+--------------------------------------+
I  Monitor-Processor       I
I  MONITR             I
+--------------------------------------+
- Implementation-Module-Code
Name Selection

Parameters: DB=MODPR3.DBF PRINT PUNCH=PSANAM.TMP EMPTY
SELECTION='TRACE-KEY="Level-1-1"' ORDER=BYTYPE

1  Monitor-Processor  PROCESS
2  Definition-Module-Code  SET
3  Implementation-Module-Code  SET
4  Monitor-Code  SET
5  Source-Code  SET
Parameters:  DB=MODPR3.DBF  FILE=PSANAM.TMP  INDEX
NOPUNCHEP- NAMES  PRINT  NOPUNCH  SMARG=5  NMARG=20
AMARG=10  BMARG=25  RNMARG=57  CMARG=1  HMARG=18
ONE-PER-LINE  COMMENT  NEW-PAGE  NONEW-LINE
NOALL-STATEMENTS  COMPLEMENTARY-STATEMENTS
NOLINE-NUMBERS  NOPRINTEOF  NODLC-COMMENT  SORT-NAME-LIST

DEFINE PROCESS Monitor-Processor;
SYNONYMS ARE: MONITR;
DESCRIPTION;
The Monitor-Processor takes a Special-File containing
Monitor-Code and transforms it into an Implementation-
Module which can be processed by the Compiler. This
involves adding an IMPORT statement, adding mutual-
exclusion controls to the exported procedures, and
converting conditions and operations on them into
SEMAPHORES and SEMAPHORE operations.
;
KEYWORDS ARE: 'Phase.II',
'process.1.1';
TRACE-KEY IS: 'Level-1',
'Level-1-1';
SUBPARTS ARE: Condition-Processing,
DEFINITION-processing,
Monitor-Setup;
PART OF: Pre-Processing;
DERIVES: Implementation-Module-Code
USING: Monitor-Code;
EMPLOYS: Definition-Module-Code;
DEFINE SET Definition-Module-Code;
SYNONYMS ARE: XXX.DEF;
DESCRIPTION;
The DEFINITION MODULE in Modula-2 defines all EXPORTED data constants, types, and variables and procedures. When another module IMPORTS something, the XXX.DEF file for that module is checked to insure that the item is EXPORTED and the definition is used for type checking.

TRACE-KEY IS: 'Level-1',
'Level-1-1',
'Level-1-2',
'Level-2-1.1';
SUBSET OF: Source-Code;
USED BY:
Compiler
TO DERIVE Symbol-File;
USED BY:
DEFINITION-processing
TO DERIVE exported-procedures-list;
EMPLOYED BY: Monitor-Processor;
DEFINE SET Implementation-Module-Code;
SYNONYMS ARE: XXX.MOD;
DESCRIPTION;
This module gives the implementation for the procedures
which are exported by the modules (as defined by XXX.D... 
The XXX.MOD is compiled separately, and the linker res...
the connections to modules which import from it.

TRACE-KEY IS:
'Level-1',
'Level-1-1',
'Level-1-2',
'Level-2-1.1';

SUBSET OF:
Source-Code;
DERIVED BY:
Condition-Processing
USING:
Monitor-Code;
DERIVED BY:
Monitor-Processor
USING:
Monitor-Code;
USED BY:
Compiler
TO DERIVE Reference-File;
USED BY:
Compiler
TO DERIVE Linker-File;
DEFINE SET Monitor-Code;
SYNONYMS ARE: XXX.MON;
DESCRIPTION;
Monitor-Code is a standard Modula-2 IMPLEMENTATION
MODULE using syntax which has been extended to include
the type `condition` and operators `signal` and `wait`
on that type. Monitor-Code must be Pre-Processed by
program MONITR prior to being compiler. The Monitor-
Processor (MONITR) will convert the XXX.MON file into
an XXX.MOD file.
;
TRACE-KEY IS: `Level-1-1`,
`Level-2-1.1`;
SUBSET OF: Special-Files;
HAS: PreemptiveScheduler-IMPORT,
MONITORSEM-declaration,
monitor-exclusion
ADDED BY Monitor-Setup;
USED BY:
Condition-Processing
TO DERIVE Implementation-Module-Code;
USED BY:
Monitor-Processor
TO DERIVE Implementation-Module-Code;
HAS: condition-type,
wait-condition,
signal-condition
MODIFIED BY Condition-Processing;
UPDATED BY: Monitor-Setup
USING: exported-procedures-list;
Formatted Problem Statement

DEFINE SET Source-Code;
DESCRIPTION;
Source-Code files are the Modula-2 statements which are suitable for compilation.
;
TRACE-KEY IS: 'Level-0',
'Level-1-1',
'Level-1-2';
SUBSETS ARE: Definition-Module-Code,
Implementation-Module-Code,
Program-Module-Code;
DERIVED BY: Program-Development
USING: Requests;
DERIVED BY: Pre-Processing
USING: Special-Files;
USED BY:
Compiler
TO DERIVE List-File;
USED BY:
Compiler-Linker
TO DERIVE Object-Code;
UPDATED BY: Program-Development
USING: Test-Run-Results;
Index

1 Compiler 3, 4(2), 6
2 Compiler-Linker 6
3 Condition-Processing 2, 4, 5(2)
4 DEFINITION-processing 2, 3
5 Definition-Module-Code 2, 3, 6
6 Implementation-Module-Code 2, 4, 5(2), 6
7 Linker-File 4
8 List-File 6
9 MONITORSEM-declaration 5
10 MONITR 2
11 Monitor-Code 2, 4(2), 5
12 Monitor-Processor 2, 3, 4, 5
13 Monitor-Setup 2, 5(2)
14 Object-Code 6
15 Pre-Processing 2, 6
16 PreemptiveScheduler-IMPORT 5
17 Program-Development 6(2)
18 Program-Module-Code 6
19 Reference-File 4
20 Requests 6
21 Source-Code 3, 4, 6
22 Special-Files 5, 6
23 Symbol-File 3
24 Test-Run-Results 6
25 XXX.DEF 3
26 XXX.MOD 4
Index

27 XXX.MON  
28 condition-type  
29 exported-procedures-list  
30 monitor-exclusion  
31 signal-condition  
32 wait-condition  

5  
5  
3, 5  
5  
5  
5
This report presents, in graphical form, the data entering or leaving a PROCESS or INTERFACE. The PROCESS or INTERFACE in question is represented as a box centered in the page. SYNONYM and a value for a user-specified ATTRIBUTE ("NODE-ATTRIBUTE") may appear in the box below the basic name of the PROCESS or INTERFACE.

Data objects entering (displayed on the left or above the box, depending upon page width) are determined from the RECEIVES, EMPLOYS, USED TO DERIVE, USED TO UPDATE, and optionally UPDATES relationships.

Data objects leaving (displayed on the right or below the box, depending upon page width) are determined from the DERIVES, GENERATES, and UPDATES relationships.

ATTRIBUTE values of the PROCESS or INTERFACE may also be displayed as the top or bottom sections of the box.

The height of the box is set by the greater of the numbers of data objects entering and leaving the PROCESS or INTERFACE.
Function Flow Data Diagram

- Definition-Module-Code
- Monitor-Code

+-----------------------------------------+
I     Monitor-Processor     I
I     MONITR     I
+-----------------------------------------+
- Implementation-Module-Code
Name Selection

Parameters: DB=MODPR3.DBF PRINT PUNCH=PSANAM.TMP EMPTY
           SELECTION='TRACE-KEY="Level-1-2"' ORDER=BYTYPE

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compiler</td>
<td>PROCESS</td>
</tr>
<tr>
<td>2</td>
<td>Linker</td>
<td>PROCESS</td>
</tr>
<tr>
<td>3</td>
<td>Definition-Module-Code</td>
<td>SET</td>
</tr>
<tr>
<td>4</td>
<td>Implementation-Module-Code</td>
<td>SET</td>
</tr>
<tr>
<td>5</td>
<td>Linker-File</td>
<td>SET</td>
</tr>
<tr>
<td>6</td>
<td>List-File</td>
<td>SET</td>
</tr>
<tr>
<td>7</td>
<td>Program-Module-Code</td>
<td>SET</td>
</tr>
<tr>
<td>8</td>
<td>Reference-File</td>
<td>SET</td>
</tr>
<tr>
<td>9</td>
<td>Source-Code</td>
<td>SET</td>
</tr>
<tr>
<td>10</td>
<td>Symbol-File</td>
<td>SET</td>
</tr>
</tbody>
</table>
Formatted Problem Statement

Parameters: DB=MODPR3.DBF FILE=PSANAM.TMP INDEX
NOPUNCHED- NAMES PRINT NOPUNCH SMARG=5 NMARG=20
AMARG=10 BMARG=25 RNMARG=57 CMARG=1 HMARG=18
ONE-PER-LINE COMMENT NEW-PAGE NONEW-LINE
NOALL-STATEMENTS COMPLEMENTARY-STATEMENTS
NOLINE- NUMBERS NOPRINT EOF NODLC- COMMENT SORT- NAME-LIST

DEFINE PROCESS Compiler;
DESCRIPTION;
The Compiler analyzes the source-code modules for errors
and produces an error listing or linker/symbol-Files. It
produces relocatable code information. A List-File is
produced even if errors do not occur.

;  
KEYWORDS ARE: 'Phase.1',
'process.2.1';
TRACE-KEY IS: 'Level-1',
'Level-1-2';
SUBPARTS ARE: Compiler-Base,
Compiler-Initialization,
Compiler-Lister,
Pass-1,
Pass-2,
Pass-3,
Pass-4,
Pass-5,
Symbol-File-Generator;
PART OF: Compiler-Linker;
DERIVES: Symbol-File
USING: Definition-Module-Code;
DERIVES: List-File
USING: Source-Code;
DERIVES: Reference-File
USING: Program-Module-Code,
Implementation-Module-Code;
DERIVES: Linker-File
USING: Program-Module-Code,
Implementation-Module-Code;
EMPLOY S: ASCII-Identifier-Table,
Interpass-File-1,
Interpass-File-2;
DEFINE PROCESS Linker;
DESCRIPTION;
The Linker links several separate modules together to build an overlay layer (program). It produces absolute (position dependent) code (core images or Object-Code).

KEYWORDS ARE: 'Phase.1',
               'process.2.2';
TRACE-KEY IS: 'Level-1',
              'Level-1-2';
PART OF:    Compiler-Linker;
DERIVES:    Object-Code
USING:      Symbol-File,
            Linker-File;
DEFINE SET Definition-Module-Code;
SYNONYMS ARE: XXX.DEF;
DESCRIPTION;
The DEFINITION MODULE in Modula-2 defines all EXPORTED
data constants, types, and variables and procedures. When another module IMPORTS something, the XXX.DEF file for that module is checked to insure that the item is EXPORTED and the definition is used for type checking.

TRACE-KEY IS: 'Level-1',
              'Level-1-1',
              'Level-1-2',
              'Level-2-1.1';
SUBSET OF:   Source-Code;
USED BY:
  Compiler
  TO DERIVE Symbol-File;
USED BY:
  DEFINITION-processing
  TO DERIVE exported-procedures-list;
EMPLOYED BY: Monitor-Processor;
FORMATTED PROBLEM STATEMENT

DEFINE SET Implementation-Module-Code;
SYNONYMS ARE: XXX.MOD;
DESCRIPTION;
This module gives the implementation for the procedures which are exported by the modules (as defined by XXX.D... The XXX.MOD is compiled separately, and the linker res... the connections to modules which import from it. ;

TRACE-KEY IS: ‘Level-1’,
‘Level-1-1’,
‘Level-1-2’,
‘Level-2-1.1’;
SUBSET OF: Source-Code;
DERIVED BY: Condition-Processing
USING: Monitor-Code;
DERIVED BY: Monitor-Processor
USING: Monitor-Code;
USED BY:
Compiler TO DERIVE Reference-File;
USED BY:
Compiler TO DERIVE Linker-File;
Formatted Problem Statement

DEFINE SET Linker-File;
TRACE-KEY IS: 'Level-1',
    'Level-1-2';
DERIVED BY: Compiler
    USING: Program-Module-Code,
        Implementation-Module-Code;
USED BY:
    Linker
    TO DERIVE Object-Code;
DEFINED SET List-File;
TRACE-KEY IS: 'Level-1',
'Level-1-2';
DERIVED BY: Compiler
USING: Source-Code;
DEFINE SET Program-Module-Code;

DESCRIPTION;
Program MODULEs are the top level of the Modula-2 hierarchy. The Source-Code for these will have a .MOD extension (like Implementation-Module-Code).

TRACE-KEY IS: 'Level-1', 'Level-1-2';

SUBSET OF: Source-Code;

USED BY:
Compiler
TO DERIVE Reference-File;

USED BY:
Compiler
TO DERIVE Linker-File;
FORMATTED PROBLEM STATEMENT

DEFINE SET Reference-File;
TRACE-KEY IS: 'Level-1',
 'Level-1-2';
DERIVED BY: Compiler
USING: Program-Module-Code,
 Implementation-Module-Code;
DEFINE SET Source-Code;
DESCRIPTION;
Source-Code files are the Modula-2 statements which are suitable for compilation.

TRACE-KEY IS: 'Level-0',
              'Level-1-1',
              'Level-1-2';

SUBSETS ARE: Definition-Module-Code,
              Implementation-Module-Code,
              Program-Module-Code;

DERIVED BY: Program-Development
USING: Requests;

DERIVED BY: Pre-Processing
USING: Special-Files;

USED BY:
   Compiler
   TO DERIVE List-File;

USED BY:
   Compiler-Linker
   TO DERIVE Object-Code;

UPDATED BY: Program-Development
USING: Test-Run-Results;
Formatted Problem Statement

DEFINE SET Symbol-File;
TRACE-KEY IS: 'Level-1',
                'Level-1-2';
DERIVED BY:    Compiler
                USING:    Definition-Module-Code;
USED BY:       Linker
                TO DERIVE Object-Code;
Index

1 ASCII-Identifier-Table 2

2 Compiler 2, 4, 5(2), 6, 7, 8(2), 9, 10, 11

3 Compiler-Base 2

4 Compiler-Initialization 2

5 Compiler-Linker 2, 3, 10

6 Compiler-Lister 2

7 Condition-Processing 5

8 DEFINITION-processing 4

9 Definition-Module-Code 2, 4, 10, 11

10 Implementation-Module-Code 2(2), 5, 6, 9, 10

11 Interpass-File-1 2

12 Interpass-File-2 2

13 Linker 3, 6, 11

14 Linker-File 2, 3, 5, 6, 8

15 List-File 2, 7, 10

16 Monitor-Code 5(2)

17 Monitor-Processor 4, 5

18 Object-Code 3, 6, 10, 11

19 Pass-1 2

20 Pass-2 2

21 Pass-3 2

22 Pass-4 2

23 Pass-5 2

24 Pre-Processing 10

25 Program-Development 10(2)
## Index

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Program-Module-Code</td>
<td>2(2), 6, 8, 9, 10</td>
</tr>
<tr>
<td>27</td>
<td>Reference-File</td>
<td>2, 5, 8, 9</td>
</tr>
<tr>
<td>28</td>
<td>Requests</td>
<td>10</td>
</tr>
<tr>
<td>29</td>
<td>Source-Code</td>
<td>2, 4, 5, 7, 8, 10</td>
</tr>
<tr>
<td>30</td>
<td>Special-Files</td>
<td>10</td>
</tr>
<tr>
<td>31</td>
<td>Symbol-File</td>
<td>2, 3, 4, 11</td>
</tr>
<tr>
<td>32</td>
<td>Symbol-File-Generator</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>Test-Run-Results</td>
<td>10</td>
</tr>
<tr>
<td>34</td>
<td>XXX.DEF</td>
<td>4</td>
</tr>
<tr>
<td>35</td>
<td>XXX.MOD</td>
<td>5</td>
</tr>
<tr>
<td>36</td>
<td>exported-procedures-list</td>
<td>4</td>
</tr>
</tbody>
</table>
Function Flow Data Diagram

Parameters:  DB=MODPR3.DBF  FILE=PSANAM.TMP
            NOTOP-ATTRIBUTE  NOBOTTOM-ATTRIBUTE  NONODE-ATTRIBUTE
            MISSING-ATTRIBUTE-VALUE='N/A'  BOX-CHARACTER='I-++++'
            SYNONYM  NOUPDATED-AS-INPUT  NODATA-OUT-DASHES  NOINDEX
            NOPUNCHEDED-NAMES  NEW-PAGE  EXPLANATION
            DATA-IN-ORDER=STANDARD  DATA-OUT-ORDER=STANDARD  NOPLOT

This report presents, in graphical form, the data entering
or leaving a PROCESS or INTERFACE. The PROCESS or INTERFACE
in question is represented as a box centered in the page.
SYNONYM and a value for a user-specified ATTRIBUTE ("NODE-
ATTRIBUTE") may appear in the box below the basic name of
the PROCESS or INTERFACE.

Data objects entering (displayed on the left or above the
box, depending upon page width) are determined from the
RECEIVES, EMPLOYS, USED TO DERIVE, USED TO UPDATE, and
optionally UPDATES relationships.

Data objects leaving (displayed on the right or below the
box, depending upon page width) are determined from the
DERIVES, GENERATES, and UPDATES relationships.

ATTRIBUTE values of the PROCESS or INTERFACE may also be
displayed as the top or bottom sections of the box.

The height of the box is set by the greater of the numbers
of data objects entering and leaving the PROCESS or INTERFACE.
Function Flow Data Diagram

- Interpass-File-1
- Interpass-File-2
- ASCII-Identifier-Table
- Definition-Module-Code
- Source-Code
- Program-Module-Code
- Implementation-Module-Code

+------------------------------------+
| I                                 |
| I                                 |
| I                                 |
| I                                 |
| I                                 |
| Compiler                          |
| I                                 |
| I                                 |
| I                                 |
| I                                 |

+------------------------------------+

- Symbol-File
- List-File
- Reference-File
- Linker-File
Function Flow Data Diagram

- Symbol-File
- Linker-File

+------------------------------------+
I                Linker                I
I
+------------------------------------+

- Object-Code
Appendix D

Level-2 PSA Formatted Problem Statement
Name Selection

Parameters:  DB=MODPR3.DBF  PRINT  PUNCH=PSANAM.TMP  EMPTY
            SELECTION='TRACE-KEY="Level-2-1.1"'  ORDER=BYTYPE

1  MONITORSEMF-declaration  ENTITY
2  PreemptiveScheduler-IMPORT  ENTITY
3  condition-type  ENTITY
4  monitor-exclusion  ENTITY
5  signal-condition  ENTITY
6  wait-condition  ENTITY
7  Condition-Processing  PROCESS
8  DEFINITION-processing  PROCESS
9  Monitor-Setup  PROCESS
10  Definition-Module-Code  SET
11  Implementation-Module-Code  SET
12  Monitor-Code  SET
13  exported-procedures-list  SET
DEFINE ENTITY MONITORSEM-declaration;
  DESCRIPTION;
  This is a variable declaration needed for the addition of mutual exclusion to a monitor. "MONITORSEM" is a reserved word for any module which will be processed by MONITR, the Monitor-Pre-Processor.

    KEYWORDS ARE: "Phase-II";
    TRACE-KEY IS: "Level-2-1.1";
    ADDED TO:
      Monitor-Code
      BY Monitor-Setup;
Formatted Problem Statement

DEFINE ENTITY PreemptiveScheduler-IMPORT;
   DESCRIPTION;
   This is the statement added to a .MON file to import the
   scheduler items needed by a monitor.
   
   KEYWORDS ARE: 'Phase-II';
   TRACE-KEY IS: 'Level-2',
                'Level-2-1.1';

   ADDED TO: Monitor-Code
              BY        Monitor-Setup;
DEFINE ENTITY condition-type;
DESCRIPTION;
The TYPE condition is an extension to Modula-2 for purposes of writing monitors for pre-processing by MONITR.

KEYWORDS ARE: 'Phase-II';
TRACE-KEY IS: 'Level-2-1.1';
MODIFIED IN:
   Monitor-Code
   BY     Condition-Processing;
DEFINE ENTITY monitor-exclusion;
DESCRIPTION;
This involves adding the Semaphore operations on
MONITORSEM which will ensure mutual exclusion for
the monitor's exported procedures.

KEYWORDS ARE: 'Phase-II';
TRACE-KEY IS: 'Level-2-1.1';
ADDED TO:
   Monitor-Code
   BY Monitor-Setup;
DEFINE ENTITY signal-condition;
   DESCRIPTION;
   A signal statement of the form 'signal(c)' where
   'c' is a 'condition' variable must be converted to
   a more complicated statement which releases a
   waiting process (if one exists) or else releases
   mutual exclusion for the monitor.

   KEYWORDS ARE: 'Phase-II';
   TRACE-KEY IS: 'Level-2-1.1';
   MODIFIED IN:
   Monitor-Code
   BY Condition-Processing;
DEFINITE ENTITY wait-condition;
   DESCRIPTION;
   A wait statement of the form 'wait(c)' where 'c' is a 'condition' variable must be converted to a more complicated statement which releases mutual exclusion for the monitor and then causes the current process to wait for a signal(c).

   KEYWORDS ARE: 'Phase-II';
   TRACE-KEY IS: 'Level-2-1.1';
   MODIFIED IN:
   Monitor-Code
   BY       Condition-Processing;
DEFINE PROCESS Condition-Processing;
DESCRIPTION;
Condition-Processing handles the conversion of type 'condition' and its operators 'signal' and 'wait' into type SEMAPHORE and operations P and V. The 'condition' type and operations on it are extensions to Modula-2 which must be handled by the Monitor-Processor before the code can be compiled.

KEYWORDS ARE: 'Phase.II',
                'process.1.1.3';
TRACE-KEY IS:  'Level-2',
                'Level-2-1.1';
PART OF:      Monitor-Processor;
DERIVES:      Implementation-Module-Code
USING:        Monitor-Code;
MODIFIES:     condition-type IN Monitor-Code;
MODIFIES:     wait-condition IN Monitor-Code;
MODIFIES:     signal-condition IN Monitor-Code;
Formulated Problem Statement

DEFINE PROCESS DEFINITION-processing;
DESCRIPTION;
This process gets the list of exported procedures from
the Definition-Module so that the Monitor-Processor
can add mutual exclusion controls to the correct parts
of the Implementation-Module.

; KEYWORDS ARE: 'Phase.II',
'process.1.1.1';
TRACE-KEY IS: 'Level-2',
'Level-2-1.1';
PART OF: Monitor-Processor;
DERIVES: exported-procedures-list
USING: Definition-Module-Code;
DEFINE PROCESS Monitor-Setup;
  DESCRIPTION;
This process adds the IMPORT statement needed to use
the scheduler, and defines the SEMAPHOREs needed to
ensure mutual exclusion for the exported procedures.
It also adds the statements needed to ensure the
mutual exclusion to each exported procedure.
;
  KEYWORDS ARE: 'Phase.II',
                 'process.1.1.2';
  TRACE-KEY IS: 'Level-2',
                 'Level-2-1.1';
  PART OF:Monitorder-Processor;
  ADDS:PreemptiveScheduler-IMPORT TO
        Monitor-Code;
  ADDS:MONITORSEM-declaration TO
        Monitor-Code;
  ADDS:monitor-exclusion TO Monitor-Code;
  UPDATES:Monitor-Code
  USING:exported-procedures-list;
DEFINITE SET Definition-Module-Code;
SYNONYMS ARE: XXX.DEF;
DESCRIPTION;
The DEFINITION MODULE in Modula-2 defines all EXPORTED data constants, types, and variables and procedures. When another module IMPORTS something, the XXX.DEF file for that module is checked to insure that the item is EXPORTED and the definition is used for type checking.

TRACE-KEY IS: 'Level-1',
            'Level-1-1',
            'Level-1-2',
            'Level-2-1.1';
SUBSET OF: Source-Code;
USED BY:
Compiler
TO DERIVE Symbol-File;
USED BY:
DEFINITION-processing
TO DERIVE exported-procedures-list;
EMPLOYED BY: Monitor-Processor;
Formatted Problem Statement

DEFINE SET Implementation-Module-Code;
SYNONYMS ARE: XXX.MOD;
DESCRIPTION;
This module gives the implementation for the procedures which are exported by the modules (as defined by XXX.D).
The XXX.MOD is compiled separately, and the linker resolves the connections to modules which import from it.

TRACE-KEY IS: 'Level-1',
'Level-1-1',
'Level-1-2',
'Level-2-1.1';
SUBSET OF: Source-Code;
DERIVED BY: Condition-Processing
USING: Monitor-Code;
DERIVED BY: Monitor-Processor
USING: Monitor-Code;
USED BY:
Compiler TO DERIVE Reference-File;
USED BY:
Compiler TO DERIVE Linker-File;
DEFINE SET Monitor-Code;
SYNONYMS ARE: XXX.MON;
DESCRIPTION;
Monitor-Code is a standard Modula-2 IMPLEMENTATION
MODULE using syntax which has been extended to include
the type 'condition' and operators 'signal' and 'wait'
on that type. Monitor-Code must be Pre-Processed by
program MONITR prior to being compiler. The Monitor-
Processor (MONITR) will convert the XXX.MON file into
an XXX.MOD file.
;
TRACE-KEY IS: 'Level-1-1',
 'Level-2-1.1';
SUBSET OF: Special-Files;
HAS: PreemptiveScheduler-IMPORT,
 MONITORSEM-declaration,
 monitor-exclusion
 ADDED BY Monitor-Setup;
USED BY:
 Condition-Processing
 TO DERIVE Implementation-Module-Code;
USED BY:
 Monitor-Processor
 TO DERIVE Implementation-Module-Code;
HAS: condition-type,
 wait-condition,
 signal-condition
 MODIFIED BY Condition-Processing;
UPDATED BY: Monitor-Setup
USING: exported-procedures-list;
DEFINE SET exported-procedures-list;
DESCRIPTION;
This is the list of procedures exported by a Monitor.

TRACE-KEY IS: 'Level-2-1.1';
DERIVED BY: DEFINITION-processing
USING: Definition-Module-Code;
USED BY:
    Monitor-Setup
    TO UPDATE Monitor-Code;
### Index

1. **Compiler** 11, 12(2)
2. **Condition-Processing** 4, 6, 7, 8, 12, 13(2)
3. **DEFINITION-processing** 9, 11, 14
4. **Definition-Module-Code** 9, 11, 14
5. **Implementation-Module-Code** 8, 12, 13(2)
6. **Linker-File** 12
7. **MONITORSEM-declaration** 2, 10, 13
8. **Monitor-Code** 2, 3, 4, 5, 6, 7, 8(4), 10(4), 12(2), 13, 14
9. **Monitor-Processor** 8, 9, 10, 11, 12, 13
10. **Monitor-Setup** 2, 3, 5, 10, 13(2), 14
11. **PreemptiveScheduler-IMPORT** 3, 10, 13
12. **Reference-File** 12
13. **Source-Code** 11, 12
14. **Special-Files** 13
15. **Symbol-File** 11
16. **XXX.DEF** 11
17. **XXX.MOD** 12
18. **XXX.MON** 13
19. **condition-type** 4, 8, 13
20. **exported-procedures-list** 9, 10, 11, 13, 14
21. **monitor-exclusion** 5, 10, 13
22. **signal-condition** 6, 8, 13
23. **wait-condition** 7, 8, 13
This report presents, in graphical form, the data entering or leaving a PROCESS or INTERFACE. The PROCESS or INTER...
in question is represented as a box centered in the page...SYNONYM and a value for a user-specified ATTRIBUTE ("NODE-ATTRIBUTE") may appear in the box below the basic name of the PROCESS or INTERFACE.

Data objects entering (displayed on the left or above the box, depending upon page width) are determined from the RECEIVES, EMPLOYS, USED TO DERIVE, USED TO UPDATE, and optionally UPDATES relationships.

Data objects leaving (displayed on the right or below the box, depending upon page width) are determined from the DERIVES, GENERATES, and UPDATES relationships.

ATTRIBUTE values of the PROCESS or INTERFACE may also be displayed as the top or bottom sections of the box.

The height of the box is set by the greater of the numbers of data objects entering and leaving the PROCESS or INTERFACE.
Function Flow Data Diagram

- Monitor-Code
  +--------------------------------------------------+
  | Condition-Processing | I |
  | I |
  +--------------------------------------------------+
- Implementation-Module-Code
Function Flow Data Diagram

- Definition-Module-Code
+--------------------------------------+
| DEFINITION-processing |
+--------------------------------------+
- exported-procedures-list
Function Flow Data Diagram

- exported-procedures-list
  +--------------------------------+
  I Monitor-Setup I
  I
  +--------------------------------+

- Monitor-Code
Name Selection

Parameters:  LANGUAGE=pds1  DB=monit2.dbf  PRINT
            PUNCH=#punch.020504  SELECTION="BASIC"  TYPE

  1  AddToken                     DATA-TYPE
  2  Name                        DATA-TYPE
  3  ParseToken                  DATA-TYPE
  4  State                       DATA-TYPE
  5  Word                        DATA-TYPE
  6  DATfile                     DEVICE
  7  DEFfile                     DEVICE
  8  MODfile                     DEVICE
  9  MONfile                     DEVICE
 10  UserTerminal                DEVICE
 11  ErrorCode                   ERROR
 12  state                       GLOBAL-VARIABLE
 13  DEFcharacters               INPUT
 14  InputUserFileName           INPUT
 15  MONcharacters               INPUT
 16  MonString                   INPUT
 17  ConvertLibrary              LIBRARY
 18  FInOut                      LIBRARY
 19  FileNames                   LIBRARY
 20  InOut                       LIBRARY
 21  MONITR.FileHandler          LIBRARY
 22  AddMonitor                  LIBRARY-ROUTINE
 23  CloseFiles                  LIBRARY-ROUTINE
 24  CommentHandler              LIBRARY-ROUTINE
 25  FInOut.CloseOutput          LIBRARY-ROUTINE
 26  FInOut.FCloseInput          LIBRARY-ROUTINE
 27  FInOut.FOpenInput           LIBRARY-ROUTINE
 28  FInOut.FOpenOutput          LIBRARY-ROUTINE
 29  FInOut.FRead                LIBRARY-ROUTINE
 30  FInOut.FReadString          LIBRARY-ROUTINE
 31  FInOut.FWriteString         LIBRARY-ROUTINE
 32  FileNames.GetFileName        LIBRARY-ROUTINE
 33  GetWord                     LIBRARY-ROUTINE
 34  InOut.WriteString           LIBRARY-ROUTINE
 35  OpenFiles                   LIBRARY-ROUTINE
 36  ReadMonTable                LIBRARY-ROUTINE
 37  Token                       LIBRARY-ROUTINE
 38  WriteMon                    LIBRARY-ROUTINE
 39  EndWord                     MANIFEST
 40  EntryString                 MANIFEST
 41  ExitString                  MANIFEST
 42  WhiteSpace                  MANIFEST
 43  MONITR.Convert               MODULE
 44  MONITR.GetDefs              MODULE
 45  ErrorMessage                OUTPUT
 46  Prompt                      OUTPUT
Name Selection

47 word.CharString OUTPUT
48 BuildCondList ROUTINE
49 ConstHandler ROUTINE
50 Convert.Main ROUTINE
51 FindProcedures ROUTINE
52 GetDefs.Main ROUTINE
53 MONITR.Main ROUTINE
54 MutexHandler ROUTINE
55 ProcedureCopy ROUTINE
56 ProcedureHandler ROUTINE
57 ReadExportList ROUTINE
58 TypeHandler ROUTINE
59 VarHandler ROUTINE
60 token-function ROUTINE-TYPE
61 MonitorPreProcessor SUBSYSTEM
62 Modula-2 SYSTEM
63 CharString VARIABLE
64 CondList VARIABLE
65 DATstring VARIABLE
66 ExpList VARIABLE
67 ExpProcList VARIABLE
68 FirstChar VARIABLE
69 HoldList VARIABLE
70 Length VARIABLE
71 MonToken VARIABLE
72 NameString VARIABLE
73 Next VARIABLE
74 ProcName VARIABLE
75 TableString VARIABLE
76 count VARIABLE
77 diskname VARIABLE
78 extension VARIABLE
79 filename VARIABLE
80 next VARIABLE
81 saveword VARIABLE
82 token VARIABLE
83 word VARIABLE
DEF DTA AddToken;  
     DESCRIPTION;  
     AddToken is used for identifying which portion of the  
     monitor is to be added by the AddMonitor routine. A  
     variable or constant of this type is passed to  
     the AddMonitor routine.  
     
     VALUE STRING 'import';  
     VALUE STRING 'type';  
     VALUE STRING 'mutex';  
     VALUE STRING 'cond';  
     VALUE STRING 'entry';  
     VALUE STRING 'exit';  
     VALUE STRING 'init';  
     
     DEFINE DTA Name;  
     DESCRIPTION;  
     Name is a linked list of names used for searching  
     for a particular name in a file.  
     
     FIELDS NameString,  
     Length,  
     Next;  
     
     DEFINE DTA ParseToken;  
     DESCRIPTION;  
     A ParseToken is an indicator of what type of word has  
     been located. There are a limited number of these  
     tokens which are valid.  
     
     VALUE STRING 'const';  
     VALUE STRING 'type';  
     VALUE STRING 'var';  
     VALUE STRING 'procedure';  
     VALUE STRING 'comment';  
     VALUE STRING 'begin';  
     VALUE STRING 'end';  
     VALUE STRING 'export';  
     VALUE STRING 'other';  
     
     DEFINE DTA State;  
     DESCRIPTION;  
     Type State is used to define the possible states for
Formatted Statements

finite state machine (the parser for MONITR.Convert is essentially a FSM). There is no good way in PDSL to define an enumerated type except by declaring it this way.

```
VALUE STRING "state1";
VALUE STRING "state2";
VALUE STRING "state3";
VALUE STRING "state4";
VALUE STRING "state5";
```

Since Modula-2 is a word oriented language, it can be parsed by words. Word separators include white-space, commas, semi-colons, parentheses, brackets and single or double quotes. The data type Word is a string of characters including white space before the actual word, and the separator which terminated the word. The first character of the word is indicated by a number, FirstChar, and the Length shows how long the word is. The character in the string after the last character of the word is the separator.

```
FIELDS CharString, FirstChar, Length;
```

```
DEFINE DEVICE DATfile;
DESCRIPTION;
This is the file containing the statements which must be added to the MONfile in order to create the MODfile.
ACCESSED SEQUENTIAL;
FORMAT IS RECORD;
DEVICE TYPE IS 'disk-file';
DEVICE USAGE input;
IN-CODE-NAME 'MONITO.DAT';
LOGICAL I/O UNIT IS 'DAT';
```

```
DEFINE DEVICE DEffile;
DESCRIPTION;
This is the DEFINITION MODULE for the user file name, filename.
ACCESSED SEQUENTIAL;
FORMAT IS STREAM;
DEVICE TYPE IS 'disk-file';
```
Formatted Statements

94 DEVICE USAGE input;
95 IN-CODE-NAME 'filename.DEF';
96 LOGICAL I/O UNIT IS 'DEF';
97
98 99 DEFINE DEVICE MODfile;
100 DESCRIPTION;
101 This is the output IMPLEMENTATION MODULE which is the
102 result of the pre-processor's action.
103 ;
104 ACCESSED SEQUENTIAL;
105 FORMAT IS STREAM;
106 DEVICE TYPE IS 'disk-file';
107 DEVICE USAGE output;
108 IN-CODE-NAME 'filename.MOD';
109 LOGICAL I/O UNIT IS 'MOD';
110
111 112 DEFINE DEVICE MONfile;
113 DESCRIPTION;
114 This is the input monitor file which will be
115 converted into an Modula-2 module.
116 ;
117 ACCESSED SEQUENTIAL;
118 FORMAT IS STREAM;
119 DEVICE TYPE IS 'disk-file';
120 DEVICE USAGE input;
121 IN-CODE-NAME 'filename.MON';
122 LOGICAL I/O UNIT IS 'MON';
123
124 125 DEFINE DEVICE UserTerminal;
126 DESCRIPTION;
127 This is the user's terminal. The user file name is
128 read from it, and any error messages are written to it.
129 ;
130 ACCESSED SEQUENTIAL;
131 FORMAT IS STREAM;
132 DEVICE TYPE IS 'TTY';
133 DEVICE USAGE input/output;
134 LOGICAL I/O UNIT IS 'standard';
135
136 137 DEFINE ERROR ErrorCode;
138 DESCRIPTION;
139 This error variable is returned by OpenFiles to the
140 MONITR.Main to indicate successful opening of all files.
141 It is a BOOLEAN which indicates an error if set to
142 FALSE when returned.
Formatted Statements

143 ;
144    ERROR IN    OpenFiles;
145
146 DEFINE GLOBAL-VARIABLE    state;
147 DESCRIPTION;
148 This is used to keep track of the current state of the
149 parser.
150 ;
151    INITIALIZE TO    'statel';
152    DATA TYPE IS    State WITH ATTRIBUTE
153      std;
154    KNOWN-BY    ConstHandler, TypeHandler, VarHandler, ProcedureHandler, Convert.Main;
155
156 DEFINE INPUT    DEFcharacters;
157 DESCRIPTION;
158 The characters read are put into a variable of type
159 Word for use by the program.
160 ;
161    READ BY    GetWord FROM    DEFfile;
162
163 DEFINE INPUT    InputUserFileName;
164 DESCRIPTION;
165 The diskname and the filename are the important parts
166 of this. The extension is supplied by the program;
167 ie, the input files are expected to have a .DEF and a
168 .MON extension, and the output will have a .MOD
169 extension. Normally, the diskname will be DK:.
170 ;
171    CONSISTS OF    diskname, filename, extension;
172    READ BY    OpenFiles FROM    UserTerminal;
173
174 DEFINE INPUT    MONcharacters;
175 DESCRIPTION;
176 The characters read are put into a variable of type
177 Word for use by the program.
178 ;
179    READ BY    GetWord FROM    MONfile;
180
181 DEFINE INPUT    MonString;
182 DESCRIPTION;
192 This is the string read in from the DATfile which
193 will be added to the MOD file.
194 ;
195 READ BY ReadMonTable FROM
196 DATfile;
197
198 DEFINE LIBRARY ConvertLibrary;
199 COLLECTION OF Token,
200 AddMonitor;
201 REFERENCED BY MONITR.Convert;
202
203 DEFINE LIBRARY FInOut;
204 DESCRIPTION;
205 This is a standard Library for Modula which does I/O
206 for files.
207 ;
208 REFERENCED BY MonitorPreProcessor,
209 Modula-2;
210
211 DEFINE LIBRARY FileNames;
212 DESCRIPTION;
213 This is a standard Library for Modula which gets a file
214 name from the user terminal. It reads it in as
215 'normal' user format (DDD:FFFFFF.EEE) and converts it
216 to 'standard' format (DDDDDDDD.FEEE).
217 ;
218 REFERENCED BY MonitorPreProcessor,
219 Modula-2;
220
221 DEFINE LIBRARY InOut;
222 DESCRIPTION;
223 This is a standard Library for Modula which does I/O
224 for the user terminal.
225 ;
226 REFERENCED BY MonitorPreProcessor,
227 Modula-2;
228
229 DEFINE LIBRARY MONITR.FileHandler;
230 DESCRIPTION;
231 The FileHandler takes care of all file opening, closing,
232 reading, and writing for the monitor pre-processor.
233
234 This is implemented as a Modula-2 module.
235
236 Several files are needed by monitor pre-processor.
237 All access to these files is done by this module,
238 which essentially hides all of the I/O operations.
239 ;
240 COLLECTION OF OpenFiles,
Formatted Statements

241 CloseFiles, 
242 GetWord, 
243 WriteMon, 
244 ReadMonTable, 
245 CommentHandler; 
246 REFERENCED BY 
247 MonitorPreProcessor, 
248 MONITR.GetDefs, 
249 MONITR.Convert; 
250 DEFINE LIBRARY-ROUTINE AddMonitor; 
251 ALGORITHM; 
252 CASE MonToken OF 
253 import, type, mutex, cond: 
254 (* the data file contains the lines to be 
255 inserted in the order in which they will be 
256 required *) 
257 REPEAT 
258 ReadMonTable(DATstring) 
259 WriteMon(DATstring) 
260 UNTIL DATstring="(****" 
261 | entry : WriteMon(EntryString) 
262 | exit : WriteMon(ExitString) 
263 | init : 
264 WriteMon(MutexInit) 
265 next := CondList 
266 WHILE next#NUL DO 
267 WriteMon("InitCond(" 
268 WriteMon(next^.Name) 
269 WriteMon(")<semicolon><rtn>"") 
270 next := next^.Next 
271 ENDWHILE 
272 ENDCASE 
273 ; 
274 DESCRIPTION; 
275 This routine is used to add required declarations and 
276 statements to the module in order to make it 
277 operational. 
278 ; 
279 NOTES; 
280 This routine is local to Convert.Main. 
281; 
282 DESIGNER; 
283 Phil Rosine; 
284 LOCAL-DATA IS next; 
285 LOCAL-DATA IS DATstring; 
286 LOCAL-MANIFEST IS EntryString, 
287 ExitString; 
288 PARAMETER MonToken PASSED-BY 
289 value;
290  PARAMETER  token PASSED-BY
291       value;
292  ROUTINE-TYPE IS  subroutine;
293  UTILIZES  ReadMonTable,
294       WriteMon;
295  COLLECTED IN  ConvertLibrary;
296  UTILIZED BY  MutexHandler,
297       Convert.Main;
298
299  DEFINE LIBRARY-ROUTINE  CloseFiles;
300  ALGORITHM;
301  FCloseInput(DAT)
302  FCloseInput(MON)
303  FCloseInput(DEF)
304  FCloseOutput(MOD)
305  ;
306  DESCRIPTION;
307  Close all files, even if there was an open error (since
308  this will not hurt anything).
309  ;
310  DESIGNER;
311   Phil Rosine;
312  ROUTINE-TYPE IS  subroutine;
313  UTILIZES  FInOut.FCloseInput,
314       FInOut.CloseOutput;
315  COLLECTED IN  MONITR.FileHandler;
316  UTILIZED BY  MONITR.Main;
317
318  DEFINE LIBRARY-ROUTINE  CommentHandler;
319  ALGORITHM;
320  AddMonitor("(*")
321  LOOP
322    GetWord(\texttt{word})
323    CASE Token\((\texttt{word})\) OF
324      comment : CommentHandler
325     | endcomment : WriteMon\((\texttt{word})\) EXIT
326    ELSE WriteMon\((\texttt{word})\)
327     END_CASE
328  END_LOOP
329  ;
330  DESCRIPTION;
331  This routine is used to copy comments from the MON file
332  to the MOD file. It is a recursive function in order
333  to handle nested comments.
334  ;
335  DESIGNER;
336   Phil Rosine;
337  ROUTINE-TYPE IS  subroutine;
338  UTILIZES  Token,
Formatted Statements

339 COLLECTED IN
340 UTILIZED BY
341
342 DEFINE LIBRARY-Routine
343 UTILIZED BY
344
345 DEFINE LIBRARY-Routine
346 UTILIZED BY
347
348 DEFINE LIBRARY-Routine
349 UTILIZED BY
350
351 DEFINE LIBRARY-Routine
352 UTILIZED BY
353
354 DEFINE LIBRARY-Routine
355 UTILIZED BY
356
357 DEFINE LIBRARY-Routine
358 UTILIZED BY
359
360 DEFINE LIBRARY-Routine
361 UTILIZED BY
362
363 DEFINE LIBRARY-Routine
364 UTILIZED BY
365
366 DEFINE LIBRARY-Routine
367 UTILIZED BY
368
369 DEFINE LIBRARY-Routine
370 UTILIZED BY
371
372 DEFINE LIBRARY-Routine
373 UTILIZED BY
374
375 WITH word DO
376 count=0
377 FRead(File, CharString[count])
378 WHILE CharString[count]=WhiteSpace DO
379 INC(count)
380 FRead(MON, CharString[count])
381 ENDWHILE
382 FirstChar := count
383 WHILE CharString[count]¹EndWord DO
384 INC(count)
385 FRead(MON, CharString[count])
386 ENDWHILE
387 Length := count - FirstChar
This routine reads a stream of characters from the MON file in such a way as to find each word in the file.

DESCRIPTION;

LOCAL-DATA IS
  count;
  WhiteSpace,
  EndWord;
LOCAL-MANIFEST IS
  word PASSED-BY
  name;
READS
  DEFcharacters FROM
  DEFfile;
READS
  MONcharacters FROM
  MONfile;
ROUTINE-TYPE IS
  subroutine;
UTILIZES
  FInOut.FRead;
COLLECTED IN
  MONITR.FileHandler;
UTILIZED BY
  CommentHandler,
  BuildCondList,
  ProcedureCopy,
  Convert.Main,
  FindProcedures,
  ReadExportList,
  MutexHandler;
DEFINE LIBRARY-ROUTINE
  InOut.WriteString;
  OpenFiles;
DEFINE LIBRARY-ROUTINE
  OpenFiles;
ALGORITHM;
FOpenInput(DAT,'DK MONITODAT')
  using InOut and FileNames,
  get user input filename 'filename'
FOpenInput(MON,'DK filenameMOD')
FOpenInput(DEF,'DK filenameDEF')
FOpenOutput(MOD,'DK filenameMOD')
IF error-on-any-open THEN
  write error message
  ErrorCode := TRUE
ENDIF
DESCRIPTION;
Prompts the user for the name of the monitor input file, and opens the required files. If an open fails, output an error message and return ErrorCode.
437 ;
438 NOTES;
439 Implemented in Modula-2, so this is a part of the
440 FileHandler module.
441 ;
442 DESIGNER;
443 Phil Rosine;
444 ERROR
445 LOCAL-DATA IS filename;
446 READS InputUserFileName FROM
447 UserTerminal;
448 ROUTINE-TYPE IS subroutine;
449 UTILIZES FInOut.FOpenInput,
450 FInOut.FOpenOutput,
451 FileNames.GetFileName,
452 InOut.WriteString;
453 WRITES Prompt TO
454 UserTerminal;
455 WRITES ErrorMessage TO
456 UserTerminal;
457 COLLECTED IN MONITR.FileHandler;
458 UTILIZED BY MONITR.Main;
459
460 DEFINE LIBRARY-ROUTINE ReadMonTable;
461 ALGORITHM;
462 FReadString(MON.TableString)
463 ;
464 DESCRIPTION;
465 This routine reads the data file containing the Modula-2
466 statements which need to be added to the monitor code
467 in order to make it a compilable module.
468 ;
469 NOTES;
470 Since this amounts to a single procedure call, the
471 implementation may actually do away with this routine.
472 ;
473 DESIGNER;
474 Phil Rosine;
475 PARAMETER TableString PASSED-BY
476 name;
477 READS MonString FROM
478 DATfile;
479 ROUTINE-TYPE IS subroutine;
480 UTILIZES FInOut.FReadString;
481 COLLECTED IN MONITR.FileHandler;
482 UTILIZED BY AddMonitor;
483 DEFINE LIBRARY-ROUTINE Token;
485 DESCRIPTION;
This function will check to see if the word matches one of the keywords in the ParseToken list, and if so return the ParseToken for that word, otherwise return ParseToken 'other'.

There are many possible algorithms which could be used, and none is detailed here.

DEFINe LIBRARY-ROUTINE WriteMon;

ALGORITHM;
FWriteString(MOD,word.CharString);

DESCRIPTION;
Writes a string to the output file (MOD).

NOTES;
Since this amounts to a single procedure call, the implementation may actually do away with this routine.

DEFINe LIBRARY-ROUTINE WriteMon;

ALGORITHM;
FWriteString(MOD,word.CharString);

DESCRIPTION;
Writes a string to the output file (MOD).

NOTES;
Since this amounts to a single procedure call, the implementation may actually do away with this routine.
Formatted Statements

535 ProcedureCopy,
536 Convert.Main,
537 MutexHandler;
538
539 DEFINE MANIFEST EndWord;
540 DESCRIPTION;
541 This is used to define what characters may define the
542 end of a word in Modula-2. Words are used as the
543 primary item for parsing a Modula-2 program, so this
544 becomes fairly important. The following may be
545 defined as word separators:
546 comma
547 semicolon
548 colon
549 WhiteSpace (see MANIFEST statement)
550 parentheses (start or end)
551 square brackets (start or end)
552 astrisk
553 slash
554 plus
555 minus
556 EOF (0C)
557 ;
558 LOCAL-MANIFEST FOR GetWord;
559
560 DEFINE MANIFEST EntryString;
561 DESCRIPTION;
562 The EntryString is the statement placed at the start
563 of each exported procedure to ensure mutual exclusion.
564 It is a Modula-2 statement coded as a string:
565 P(MONITORSEM)<semicolon>
566 ;
567 LOCAL-MANIFEST FOR AddMonitor;
568
569 DEFINE MANIFEST ExitString;
570 DESCRIPTION;
571 The ExitString is the statement placed at the end of
572 each exported procedure to ensure mutual exclusion.
573 It is a Modula-2 statement coded as a string:
574 <semicolon>IF MONITORCOND.cc > 0
575 THEN V(MONITORCOND.c)
576 ELSE V(MONITORSEM)
577 END
578 ;
579 LOCAL-MANIFEST FOR AddMonitor;
580
581 DEFINE MANIFEST WhiteSpace;
582 DESCRIPTION;
583 WhiteSpace is used in order to skip over excess blank
Formatted Statements

584 space between words. It includes the following:
585 space
586 tab
587 carriage return
588 linefeed
589 ;
590 LOCAL-MANIFEST FOR GetWord;
591
592 DEFINE MODULE MONITR.Convert;
593 DESCRIPTION;
594 This module reads in the monitor file and adds the
595 neccessary statements to make it a compilable,
596 working Modula-2 module which acts as a monitor.
597 ;
598 IMPLEMENTATION;
599 This is implemented as a Modula-2 module.
600 ;
601 DESIGNER;
602 Phil Rosine;
603 PURPOSE;
604 Statements are added to import the SEMAPHORE type and
605 its operations from PreemptiveScheduler, to define
606 the mutual exclusion variables, to define the type
607 CONDITION and its operations, to ensure mutual
608 exclusion for the exported procedures, and to
609 initialize the new variables.
610 ;
611 INTERFACE ROUTINE Convert.Main;
612 REFERENCES ConvertLibrary,
613 MONITR.FileHandler;
614 ROUTINE BuildCondList,
615 ConstHandler,
616 TypeHandler,
617 VarHandler,
618 ProcedureHandler,
619 ProcedureCopy,
620 MutexHandler;
621 INVOKED BY MONITR.Main THROUGH
622 Convert.Main;
623 Module IN MonitorPreProcessor;
624
625 DEFINE MODULE MONITR.GetDefs;
626 DESCRIPTION;
627 This module handles reading the DEF file and finding
628 the exported procedure names.
629 ;
630 IMPLEMENTATION;
631 This is implemented as a Modula-2 module.
632 ;
In order to implement a monitor, each exported procedure must have an entry and an exit routine inserted to insure mutual exclusion for user processes. A list of these exported routines is required in order to insert the entry and exit statements. GetDefs locates the EXPORT list in the DEFINITION MODULE and returns a list of which exported items are procedures.

INTERFACE ROUTINE GetDefs.Main;
REFERENCES MONITR.FileHandler;
ROUTINE ReadExportList,
FindProcedures;
INVOKED BY MONITR.Main THROUGH GetDefs.Main;
MODULE IN MonitorPreProcessor;

DEFINE OUTPUT ErrorMessage;
DESCRIPTION;
Error messages are written to indicate a failure to open a file. This may be due to non-existence of the DAT, DEF, or MON files, or inability to create the MOD file.

WRITE BY OpenFiles TO UserTerminal;

DEFINE OUTPUT Prompt;
DESCRIPTION;
The prompt is used to request a file name from the user.

WRITE BY OpenFiles TO UserTerminal;

DEFINE OUTPUT word.CharString;
DESCRIPTION;
This is a CharString of a variable `word' of type Word. It is a null terminated string of characters.

WRITE BY WriteMod TO MODfile;

DEFINE ROUTINE BuildCondList;

ALGORITHM;
LOOP
GetWord(word)
Formatted Statements

682 CASE Token(word) OF
683    procedure : EXIT
684    | comment : CommentHandler
685    | condition : copy HoldList to end of CondList
686    ELSE add word to HoldList
687 END CASE
688 IF word.CharString[FirstChar+Length]=semicolon THEN
689   empty HoldList ENDIF
690 ENDLOOP
691 ;
692 DESCRIPTION;
693 This routine builds the list of variables declared as
694 type CONDITION which is required for adding the
695 initialization statements.
696 ;
697 DESIGNER;
698 Phil Rosine;
699 LOCAL-DATA IS HoldList;
700 LOCAL-DATA IS word;
701 PARAMETER word PASSED-BY name;
702 PARAMETER CondList PASSED-BY name;
703 ROUTINE-TYPE IS subroutine;
704 UTILIZES CommentHandler,
705 GetWord,
706 Token,
707 WriteMon;
708 CALLED-BY Convert.Main;
709 ROUTINE IN MONITR.Convert;
710 DEFINE ROUTINE ConstHandler;
711 ALGORITHM;
712 AddMonitor (*IMPORT*)
713 WriteMon(word)
714 state := state2
715 ;
716 DESCRIPTION;
717 This routine adds the IMPORT statement to the module
718 when a CONST declaration is found.
719 ;
720 NOTES;
721 This routine is local to Convert.Main.
722 ;
723 DESIGNER;
724 Phil Rosine;
725 KNOWS-OF state;
726 PARAMETER word PASSED-BY value;
ROUTINE-TYPE IS subroutine;
UTILIZES WriteMon;
CALLED-BY Convert.Main;
ROUTINE IN MONITR.Convert;

DEFINE ROUTINE Convert.Main;
ALGORITHM;
CondList := NUL
state := statel
LOOP
    GetWord(word)
    CASE state OF
        state1,state2,state3 :
            CASE Token(word) OF
                const : ConstHandler(state)
                | type : TypeHandler(state)
                | var : VarHandler(state)
                | procedure : ProcedureHandler(state)
                | comment : CommentHandler
                ELSE
                    WriteMon(word)
                    ENDCLASS
            ENDCLASS
        state4 :
            BuildCondList(CondList,word)
            CASE Token(word) OF
                procedure : ProcedureHandler(state)
                | comment : CommentHandler
                ELSE WriteMon(word)
                ENDCLASS
            ENDCLASS
        state5 :
            CASE Token(word) OF
                procedure : MutexHandler
                | begin :
                    WriteMon(word)
                    AddMonitor (*Mutual Exclusion setup*)
                    AddMonitor(CondList)
                    WHILE NOT EOF DO
                        GetWord(word)
                        WriteMon(word)
                        ENDFWHILE
                    EXIT
                    | comment : CommentHandler
                    ELSE
                    WriteMon(word)
                    ENDCLASS
                ENDCLASS
            ENDCLASS
        ENDCLASS
    ENDCLASS
ENDLOOP

; 
DESCRIPTION;
This is the primary parser for the pre-processor. It locates the keywords and initiates appropriate actions.

NOTES;
The ROUTINES ConstHandler, TypeHandler, VarHandler, MutexHandler, ProcedureHandler, and ProcedureCopy are local routines to this routine.

DESIGNER;
Phil Rosine;

CALLS
BuildCondList,
ConstHandler,
MutexHandler,
TypeHandler,
VarHandler,
ProcedureHandler;

KNOWS-OF
state;

LOCAL-DATA IS
word;

ROUTINE-TYPE IS
subroutine;

UTILIZES
AddMonitor,
CommentHandler,
GetWord,
Token,
WriteMon;

INTERFACE ROUTINE FOR
MONITR.Convert;

DEFINE ROUTINE
FindProcedures;

ALGORITHM;
ExpProcList := NUL
REPEAT
GetWord(word)
CASE Token(word) OF
procedure:
GetWord(word)
IF word IN ExpList THEN
add word to ExpProcList
ENDIF
| comment : CommentHandler
ELSE (*nothing*)
ENDCASE
UNTIL word.CharString[FirstChar+Length]=EOF

DESCRIPTION;
This routine finds the exported names which are procedures.

DESIGNER;
Phil Rosine;
LOCAL-DATA IS
PARAMETER ExpList PASSED-BY
name;
PARAMETER ExpProcList PASSED-BY
name;
ROUTINE-TYPE IS subroutine;
UTILIZES CommentHandler,
GetWord,
Token;
CALLED-BY GetDefs.Main;
ROUTINE IN MONITR.GetDefs;

DEFINE ROUTINE GetDefs.Main;
ALGORITHM;
ReadExportList(ExpList)
FindProcedures(ExpList,ExpProcList)
;
DESCRIPTION;
This routine gets the list of exported procedures from
the DEF file.
;
DESIGNER;
Phil Rosine;
CALLS FindProcedures,
ReadExportList;
PARAMETER ExpProcList PASSED-BY
name;
ROUTINE-TYPE IS subroutine;
INTERFACE ROUTINE FOR
MONITR.GetDefs;

DEFINE ROUTINE MONITR.Main;
ALGORITHM;
OpenFiles
IF NOT ErrorCode THEN
GetDefs(ExpProcList)
Convert(ExpProcList)
ENDIF
CloseFiles
;
DESIGNER;
Phil Rosine;
INVOKES MONITR.Convert THROUGH
Convert.Main;
INVOKES MONITR.GetDefs THROUGH
GetDefs.Main;
LOCAL-DATA IS ExpProcList;
ROUTINE-TYPE IS subroutine;
UTILIZES CloseFiles,
Formatted Statements

878     879     880
881     882     883     884
885     886     887     888
889     890     891     892
893     894     895     896
897     898     899     900
901     902     903     904
905     906     907     908
909     910     911     912
913     914     915     916
917     918     919     920
921     922     923     924
925     926

878 OpenFiles;
879 MAIN ROUTINE FOR MonitorPreProcessor;
880
881 DEFINE ROUTINE MutexHandler;
882 ALGORITHM;
883 WriteMon(word)
884 GetWord(word)
885 IF Token(word)=comment THEN CommentHandler
886 ELSE WriteMon(word) ENDIF
887 IF word IN ExpProcList THEN
888     ProcName := word
889     LOOP
890     GetWord(word)
891     CASE Token(word) OF
892     begin :
893         WriteMon(word)
894         AddMonitor (*ENTRY*)
895         saveword := word
896         GetWord(word)
897         IF word=ProcName
898         THEN AddMonitor (*EXIT*) ENDIF
899         WriteMon(saveword)
900         writeMon(word)
901     | procedure :
902         GetWord(word)
903         ProcedureCopy(word)
904     | "(*) : CommentHandler
905     ELSE
906         WriteMon(word)
907     ENDCASE
908     ENDLOOP
909     ELSE
910     ProcedureCopy(ProcName)
911     ENDFIF
912
913 DESCRIPTION;
914 This routine adds the entry and exit statements to
915 exported procedures in order to ensure mutual exclusion.
916 NOTES;
917 This routine is local to Convert.Main.
920;
921 DESIGNER;
922 Phil Rosine;
923 CALLS ProcedureCopy;
924 LOCAL-DATA IS ProcName;
925 LOCAL-DATA IS saveword;
926 PARAMETER word PASSED-BY
Sem Version 1.3R3 Apr 28, 1984 11:34:54 Page 22
Univ-of-Montana-VAX-11/750

Formatted Statements

927 name;
928 ROUTINE-TYPE IS subroutine;
929 UTILIZES AddMonitor,
930 GetWord,
931 Token,
932 WriteMon;
933 CALLED-BY Convert.Main;
934 ROUTINE IN MONITR.Convert;
935
936 DEFINE ROUTINE ProcedureCopy;
937 ALGORITHM;
938 WriteMon(word)
939 ProcName := word
940 LOOP
941 GetWord(word)
942 CASE Token(word) OF
943 end :
944 WriteMon(word)
945 GetWord(word)
946 WriteMon(word)
947 IF word=ProcName THEN EXIT ENDF
948 | comment : CommentHandler
949 ELSE WriteMon(word)
950 ENDCASE
951 ENDLOOP
952 ;
953 DESCRIPTION;
954 This procedure is used to copy procedures from the MON
955 file to the MOD file. This is used both for nested
956 procedures within an exported procedure, and for
957 copying non-exported procedures.
958 ;
959 NOTES;
960 This routine is local to Convert.Main.
961 ;
962 DESIGNER;
963 Phil Rosine;
964 LOCAL-DATA IS ProcName;
965 PARAMETER word PASSED-BY
966 value;
967 ROUTINE-TYPE IS subroutine;
968 UTILIZES CommentHandler,
969 GetWord,
970 Token,
971 WriteMon;
972 CALLED-BY MutexHandler;
973 ROUTINE IN MONITR.Convert;
974
975 DEFINE ROUTINE ProcedureHandler;
Algorithm;
IF (state=state1) OR (state=state2) THEN
  AddMonitor (*IMPORT*)
  AddMonitor (*Mutual Exclusion VARs*)
ENDIF
WriteMon(word)
state := state5

Description;
This is called when the first PROCEDURE declaration is found. It adds the IMPORT and mutual exclusion variables if required. The procedure declarations and TYPE declarations for handling CONDITIONS are not required since if this procedure is called there have been no VAR declarations and therefore no CONDITIONS.

Notes;
This routine is local to Convert.Main.

Designer;
Phil Rosine;

Known-Of;

Routine-Type is subroutine;
Utilizes WriteMon;
Called-By Convert.Main;
Routine in MONITR.Convert;

Define Routine
ReadExportList;
Algorithm;
Loop
GetWord(word)
CASE Token(word) OF
  comment : CommentHandler
  export :
    WHILE word.CharString[FirstChar+Length]#semicolon
      add word to ExpList
    ENDWHILE
  ELSE (* nothing *)
  ENDCASE
ENDLoop

Description;
This routine finds the EXPORT list in the DEF files and places the names from it into a list.
1025 DESIGNER;  
1026 Phil Rosine;  
1027 LOCAL-DATA IS word;  
1028 PARAMETER ExpList PASSED-BY  
1029 name;  
1030 ROUTINE-TYPE IS subroutine;  
1031 UTILIZES GetWord, Token;  
1032 CALLED-BY GetDefs.Main;  
1033 ROUTINE IN MONITR.GetDefs;  
1034  
1035 DEFINERoutine  
1036 TypeHandler;  
1037 ALGORITHM;  
1038 IF state=state1 THEN  
1039 AddMonitor (*IMPORT*)  
1040 ENDIF  
1041 WriteMon(word)  
1042 AddMonitor (*TYPE CONDITION*)  
1043 state := state3  
1044 ;  
1045 DESCRIPTION;  
1046 This routine adds IMPORT statement if required, and  
1047 then adds the TYPE declaration for CONDITION. It is  
1048 called when a TYPE declaration is found.  
1049 ;  
1050 NOTES;  
1051 This routine is local to Convert.Main.  
1052 ;  
1053 DESIGNER;  
1054 Phil Rosine;  
1055 KNOWS-OF state;  
1056 PARAMETER word PASSED-BY  
1057 value;  
1058 ROUTINE-TYPE IS subroutine;  
1059 UTILIZES WriteMon;  
1060 CALLED-BY Convert.Main;  
1061 ROUTINE IN MONITR.Convert;  
1062  
1063 DEFINERoutine  
1064 VarHandler;  
1065 ALGORITHM;  
1066 IF state=state1 THEN  
1067 AddMonitor (*IMPORT*)  
1068 ENDIF  
1069 IF state=state2 THEN  
1070 AddMonitor (*TYPE CONDITION*)  
1071 ENDIF  
1072 WriteMon(word)  
1073 AddMonitor (*Mutual Exclusion VARS*)
Formatted Statements

1074 state := state4
1075 ;
1076 DESCRIPTION;
1077 This routine is called when a VAR declaration is found.
1078 It adds the IMPORT and TYPE statements if required, and
1079 then adds the variable declarations needed for mutual
1080 exclusion handling.
1081 ;
1082 NOTES;
1083 This routine is local to Convert.Main.
1084 ;
1085 DESIGNER;
1086 Phil Rosine;
1087 KNOWS-OF state;
1088 PARAMETER word PASSED-BY value;
1089 ROUTINE-TYPE IS subroutine;
1090 UTILIZES WriteMon;
1091 CALLED-BY Convert.Main;
1092 ROUTINE IN MONITR.Convert;
1093
1094 DEFINE ROUTINE-TYPE token-function;
1095 DESCRIPTION;
1096 This routine type is a function which returns a value
1097 of type ParseToken.
1098 ;
1099 ;
1100 DEFINE SUBSYSTEM MonitorPreProcessor;
1101 SYNONYMS ARE MONITR;
1102 DESCRIPTION;
1103 The monitor pre-processor accepts Modula-2 code with
1104 some syntax extensions (type CONDITION and operations
1105 SIGNAL(C) and WAIT(C)). This code is transformed
1106 into a compilable Modula-2 IMPLEMENTATION MODULE.
1107 ;
1108 DESIGNER;
1109 Phil Rosine;
1110 USAGE-INFORMATION;
1111 This is a standard Modula-2 program, runnable on a
1112 PDP-11 under the RT-11 operating system by use of the
1113 MODULA program.
1114
1115 It requires as input from the user the name of a file
1116 containing the monitor to be converted. This file will
1117 normally have a .MON file extension. Also available
1118 must be a .DEF file with the same filename, and a
1119 .MOD file of that name will be produced as output.
1120 The .MOD file produced must then be run through the
1121 Modula-2 compiler in a normal fashion and linked with
1123 a program. The pre-processor will use a data file
1124 containing the statements to be inserted into the
1125 monitor code as input during the conversion process.
1126 ;
1127 MAIN ROUTINE IS MONITR.Main;
1128 MODULE MONITR.Convert,
1129 MONITR.GetDefs;
1130 REFERENCES FInOut,
1131 FileNames,
1132 InOut,
1133 MONITR.FileHandler;
1134 SUBSYSTEM FOR Modula-2;
1135
1136 DEFINE SYSTEM Modula-2;
1137 DESCRIPTION;
1138 Modula-2 is a combination of programming language,
1139 compiler, linker, and run-time-system.
1140 ;
1141 DESIGNER;
1142 Niklaus Wirth;
1143 USAGE-INFORMATION;
1144 Runs under RT-11 on PDP-11/40 and PDP-11/03 computers
1145 (includes PDP-11/23 systems like we have).
1146 ;
1147 REFERENCES FInOut,
1148 FileNames,
1149 InOut;
1150 SUBSYSTEM MonitorPreProcessor;
1151
1152 DEFINE VARIABLE CharString;
1153 DESCRIPTION;
1154 CharString contains the data read into a variable of
1155 type Word. From CharString[0] to CharString[FirstChar-1]
1156 the string contains any leading whitespace (separating
1157 two words). CharString[FirstChar+Length] contains the
1158 endword character which indicated the end of the word.
1159 The end of the data is indicated by a NULL (0C) (unless
1160 the string is of maximum length).
1161 ;
1162 LENGTH IS 60;
1163 DATA TYPE IS string WITH ATTRIBUTE
1164 std;
1165
1166 DEFINE VARIABLE CondList;
1167 DESCRIPTION;
1168 This is a pointer to type Name.
1169 ;
1170 POINTER FOR NameString;
1171 DATA TYPE IS Name WITH ATTRIBUTE
1172 std;
1173 PARAMETER FOR BuildCondList PASSED-BY
1174 name;
1175
1176 DEFINE VARIABLE DATstring;
1177 LENGTH IS 132;
1178 DATA TYPE IS string WITH ATTRIBUTE
1179 std;
1180 LOCAL-DATA FOR AddMonitor;
1181
1182 DEFINE VARIABLE ExpList;
1183 DESCRIPTION;
1184 This is a pointer to type Name.
1185 ;
1186 POINTER FOR NameString;
1187 DATA TYPE IS Name WITH ATTRIBUTE
1188 std;
1189 PARAMETER FOR FindProcedures PASSED-BY
1190 name;
1191 PARAMETER FOR ReadExportList PASSED-BY
1192 name;
1193
1194 DEFINE VARIABLE ExpProcList;
1195 DESCRIPTION;
1196 This is a pointer to type Name.
1197 ;
1198 POINTER FOR NameString;
1199 DATA TYPE IS Name WITH ATTRIBUTE
1200 std;
1201 LOCAL-DATA FOR MONITR.Main;
1202 PARAMETER FOR FindProcedures PASSED-BY
1203 name;
1204 PARAMETER FOR GetDefs.Main PASSED-BY
1205 name;
1206
1207 DEFINE VARIABLE FirstChar;
1208 DESCRIPTION;
1209 FirstChar is the location of the first character of the
1210 actual word in the CharString portion of a variable of
1211 type Word.
1212 ;
1213 DATA TYPE IS integer WITH ATTRIBUTE
1214 std;
1215
1216 DEFINE VARIABLE HoldList;
1217 LOCAL-DATA FOR BuildCondList;
1218
1219 DEFINE VARIABLE Length;
1220 DESCRIPTION;
Indicates the length of the actual word within CharString of a variable of type Word (starting from FirstChar). Also used to keep the length of a NameString. This variable is a field within both record types NameString and Word.

DATA TYPE IS integer WITH ATTRIBUTE std;

DEFINE VARIABLE MonToken;
PARAMETER FOR AddMonitor PASSED-BY value;

DEFINE VARIABLE NameString;
DESCRIPTION;
The NameString is the actual name of a PROCEDURE orEXPORTED item as a part of a Name.

LENGTH IS 40;
DATA TYPE IS string WITH ATTRIBUTE std;
POINTED-TO BY CondList,
ExpList,
ExpProcList,
Next;

DEFINE VARIABLE Next;
DESCRIPTION;
Points to the next item in a NameList, which is a linked list of items of type Name.

POINTER FOR NameString;

DEFINE VARIABLE ProcName;
LOCAL-DATA FOR MutexHandler;
LOCAL-DATA FOR ProcedureCopy;

DEFINE VARIABLE TableString;
LENGTH IS 132;
DATA TYPE IS string WITH ATTRIBUTE std;
PARAMETER FOR ReadMonTable PASSED-BY name;

DEFINE VARIABLE count;
DATA TYPE IS integer WITH ATTRIBUTE std;
LOCAL-DATA FOR GetWord;
DEFINE VARIABLE diskname;
DESCRIPTION;
A diskname is the logical name for a disk storage unit on the RT-11 operating system. It consists of 1-3 characters followed by a colon.

LENGTH IS 4;
DATA TYPE IS string WITH ATTRIBUTE std;
CONTAINED IN InputUserFileName;

DEFINE VARIABLE extension;
DESCRIPTION;
An extension is the final portion of a file name which can be used to differentiate between several files of the same filename. It can be 0-3 characters which are separated from the filename itself by a period (dot).

LENGTH IS 3;
DATA TYPE IS string WITH ATTRIBUTE std;
CONTAINED IN InputUserFileName;

DEFINE VARIABLE filename;
LENGTH IS 12;
DATA TYPE IS string WITH ATTRIBUTE std;
CONTAINED IN InputUserFileName;
LOCAL-DATA FOR OpenFiles;

DEFINE VARIABLE next;
LOCAL-DATA FOR AddMonitor;

DEFINE VARIABLE saveword;
LOCAL-DATA FOR MutexHandler;

DEFINE VARIABLE token;
DATA TYPE IS AddToken WITH ATTRIBUTE std;
PARAMETER FOR AddMonitor PASSED-BY value;

DEFINE VARIABLE word;
DATA TYPE IS Word WITH ATTRIBUTE std;
LOCAL-DATA FOR BuildCondList;
LOCAL-DATA FOR Convert.Main;
LOCAL-DATA FOR FindProcedures;
LOCAL-DATA FOR ReadExportList;
Formatted Statements

1319  PARAMETER FOR               GetWord PASSED-BY
      name;
1320  PARAMETER FOR               Token PASSED-BY
      name;
1321  PARAMETER FOR               WriteMon PASSED-BY
      name;
1322  PARAMETER FOR               BuildCondList PASSED-BY
      name;
1323  PARAMETER FOR               ConstHandler PASSED-BY
      name;
1324  PARAMETER FOR               MutexHandler PASSED-BY
      name;
1325  PARAMETER FOR               ProcedureCopy PASSED-BY
      value;
1326  PARAMETER FOR               ProcedureHandler PASSED-BY
      value;
1327  PARAMETER FOR               TypeHandler PASSED-BY
      value;
1328  PARAMETER FOR               VarHandler PASSED-BY
      value;
1329  PARAMETER FOR               value;
1330  PARAMETER FOR               value;
1331  PARAMETER FOR               value;
1332  PARAMETER FOR               value;
1333  PARAMETER FOR               value;
1334  PARAMETER FOR               value;
1335  PARAMETER FOR               value;
1336  PARAMETER FOR               value;
1337  PARAMETER FOR               value;
1338  PARAMETER FOR               value;

1338 lines printed. 492 statements printed.
REFERENCES
An Annotated Bibliography

[Ben-Ari, 1982].
Ben-Ari, M. Principles of Concurrent Programming. Prentice-Hall International, Inc., 1982. This is an general overview of the problems and solutions to concurrency, with emphasis on semaphores, monitors, and rendezvous. A number of 'classic problems' are presented and solutions using several constructs are given.

[Basili, 1981].

[BrinchHansen, 1977a].

[BrinchHansen, 1977b].
Brinch Hansen, Per. "Experience with Modular Concurrent Programming." IEEE Transactions on Software
Engineering. Vol. SE-3, No. 2, March 1977. pp 156-159. This is a summary of the author's initial experience with Concurrent Pascal.

[BrinchHansen, 1978].
Brinch Hansen, Per. "Distributed Processes: A Concurrent Programming Concept." Communications of the ACM. Association for Computing Machinery, August 1978, Volume 21 Number 11, pp. 934-941. This discusses a language concept for concurrency without common variables, with interprocess communication being carried out via procedure calls and guarded regions.

[Campbell]

[DeMarco, 1979].

[Dijkstra, 1971].

[Dijkstra, 1975].
Dijkstra, Edsger W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". Communications of the ACM. Association for Computing Machinery, August 1975, pp. 453-457. This paper introduces the concept of guarded commands for alternative and repetitive constructs which allow nondeterministic program development.

[Dobrowolski]
Dobrowolski, Gregory. "Eventcounts and Sequencers".

[Eventoff, 1980].
Efficiency Difference?". Communications of the ACM. Association for Computing Machinery, 1980. pp 156-165. This is a discussion of the differences between the Ada rendezvous and and monitor. Particular emphasis is on the experimental efficiency measurements based upon two particular implementations. Some detail of the implementations is given.

[Hoare, 1974].

[Hoare, 1978].
Hoare, C.A.R. "Communicating Sequential Processes". Communications of the ACM. Association for Computing Machinery, August 1978, pp. 666-677. This paper combines the premise that input and output are the basic primitives of programming with Dijkstra's [1975] guarded command constructs to introduce a method of inter-process communication and synchronization based upon message passing.

[Geissmann, 1980].

[Geissmann, 1981A].

[Geissmann, 1981B].

[Jensen, 1978].
Jensen, K., and N. Wirth. PASCAL User Manual and Report. Springer-Verlag, New York, 1978. This is the original language definition and tutorial for
Pascal.


[Taylor,1983] Taylor, Richard N. "A General-Purpose Algorithm for Analyzing Concurrent Programs." Communications of the ACM. Vol 26, No 5, May 1983, pp 362-376. This paper presents an algorithm for doing static analysis of some aspects of concurrent programs. The paper is primarily focused on Ada, but is applicable to other concurrent programming languages.


