The MODUS Quarterly

Issue # 8

May 1987

Modula-2 News for MODUS, the Modula-2 Users Association.

CONTENT

Directors of MODUS, the Modula-2 Users Association:

Randy Bush
Oregon Software
6915 South West Macadam
Portland, OR 97219
(503) 245-2202

Tom DeMarco
Atlantic Systems Guild
353 West 12th Street
New York, NY 10014
(212) 620-4282

Jean-Louis Dewez
Laboratoire de Micro Informatique
Conserveratoire NAM
2, Rue Conte
F-75003 Paris
(01) 42 71 24 14

Svend Erik Knudsen
Institut fuer Informatik
ETH Zuerich
CH-8092 Zuerich
(01) 256 3487

Heinz Waldburger
ERDIS SA
CH 1800 Vevey 2
(021) 52 61 71

Administration and membership:

USA: George Symons
MODUS
PO Box 51778
Palo Alto, CA 94303
(415) 322-0547

Europe: Aline Sigrist
MODUS Secretary
ERDIS SA
P. O. Box 35
CH 1800 Vevey 2

Editor, Modula-2 News:

Richard Karpinski
6521 Raymond Street
Oakland, CA 94609
Weekdays (415) 476-4529 (11-7 pm)
Anytime (415) 658-3797 (ans. mach.)
TeleMail M2News or RKarpinski
BITNET dick@ucsfcca Internet dick@cca.ucsf.edu
Compuserve 70215,1277 USENET ...!ucbvax!ucsfcgl!cca.ucsf!dick

>> Problems? Missing an issue? <<

Contact your membership coordinator (see above).

Publisher:

George Symons (see above)

Putative publication schedule:

Deadline	Issue
15 Jan	Feb
15 Apr	May
15 Jul	Aug
15 Oct	Nov

Submissions for publication:

Send CAMERA READY copy to the editor.
Dot matrix copy is often unacceptable.
Machine readable copy is preferred: 60 lines, 70/84 characters.

TeleMail address: M2News

Please indicate that publication of your submission is permitted.
Correspondence not for publication should be PROMINENTLY so marked.

Modus Meeting Announcement & Call for Participation

June 15 & 16, 1987 / San Francisco, California

The Modula-2 Users' Association will host a semi-formal technical meeting at the Seven Hills Center in San Francisco, California. The Seven Hills Center is a conference facility located on the campus of San Francisco State University (SFSU) in southwestern San Francisco.

Call for Participation

Technical papers and presentations are solicited from the Modula-2 community; you need not be a member of Modus. Suggested topics include: language and library issues, educational uses, industrial uses, new implementations, real world uses, and the progress of standardization.

By the 22nd of May 1987, please send an abstract of your presentation or paper to:

Program Coordinator
MODUS Meeting
P.O. Box 51778
Palo Alto, California USA - 94303

If you will need audio-visual equipment, let us know your requirements. If you wish your paper to be published in *MODUS Quarterly*, the Editor must receive a copy of the completed paper. If you wish to discuss your presentation or paper ideas, contact Stan Osborne at (415) 341-1768, (UUCP: ...!ucbvax!dual!dbi!stan).

If you want to make technical demonstrations, additional meeting rooms can be rented from the Seven Hills Center. Please contact us with the details of your space and power requirements, if you are interested in making a commercial or non-commercial technical demonstration.

Registration Information

In addition to the technical sessions a light breakfast, lunch, and afternoon refreshments will be provided on each day. An informal reception is scheduled for Monday evening. The registration fee for one or two days includes the technical sessions and food. All people registering will be admitted to the Monday evening reception.

Registration by Mail:

Both days: $55.00; One day: $35.00

On Site Registration:

Both days: $70.00; One day: $50.00

To qualify for the "Registration by Mail" fee your check must be received before the 8th of June, 1987. Send your registration request along with a check to:

MODUS Meeting Registration
P.O. Box 51778
Palo Alto, California USA - 94303

The Seven Hills Center is located at 800 Font Boulevard on the western part of the 100 acre SFSU Campus, near Lake Merced in the southwestern corner of San Francisco. It is one mile from the Pacific Ocean; 25 minutes by freeway or public transportation from downtown San Francisco or the San Francisco International Airport. If you need help in finding the SFSU Campus or the Seven Hills Center, you may phone (415) 469-1067 for assistance.

Places to Stay in San Francisco

In addition to the many first class Hotels and Motels located in downtown San Francisco, Fisherman's Warf, and Lombard Street (Highway 101) there are a number of Motels closer to the University. A partial list of good nearby Motels follows:

Great Highway Motor Inn, 2180 Great Highway (at the southwestern corner of Golden Gate Park). Discount for SFSU patrons. Prices range from $40.- and up. Telephone: (415) 731-6644.

Mission Bell Motel, 6843 Mission Street in Daly City (a quaint antique, kept clean and low priced). Prices range from $25.- and up. Telephone: (415) 755-6161.

Ocean Park Motel, 2690 - 46th Avenue (one block from the zoo). Prices range from $50.- and up. Telephone: (415) 566-7020.

Roberts Motel, 2828 Sloat Boulevard (across from the zoo). Prices range from $41.- and up. Telephone: (415) 564-2610.

Seal Rock Inn, 48th Avenue at Point Lobos (above the Cliff House). Prices range from $56.- and up. Telephone: (415) 752-8000.

Sunset Motel, 821 Taraval Street (closest to campus). Prices range from $41.- and up. Telephone: (415) 564-3635.

Modus Meeting Announcement and Call for Participation

June 15 & 16, 1987

San Francisco, California

The Modula-2 Users' Association will host a semi-formal technical meeting at the Seven Hills Center in San Francisco, California. The Seven Hills Center is located at 800 Font Boulevard on the western part of the 100 acre San Francisco State University Campus, near Lake Merced in the southwestern corner of San Francisco.

Technical papers and presentations are solicited from the Modula-2 community; you need not be a member of Modus. Suggested topics include: language and library issues, educational uses, industrial uses, new implementations, real world uses, and the progress of standardization. Some members of the ISO Working Group standardizing Modula-2 will report on their efforts and particular interests.

Before the 22nd of May 1987, please send an abstract of your presentation or paper to:

Program Coordinator, MODUS Meeting
P.O. Box 51778, Palo Alto, California USA - 94303

Or contact Stan Osborne at (415) 341-1768, (UUCP: ...!ucbvax!dual!dbi!stan).

If you wish to attend this event, complete the registration application below and mail it with a check for the right amount to:

MODUS Meeting Registration
P.O. Box 51778, Palo Alto, California USA - 94303

You will be sent a confirmation receipt, a San Francisco map and information about motels close to the Seven Hills Center.

MODUS Meeting Registration Form, June 15 & 16, 1987

Name : _____

Affiliation : _____

Address : _____

Address : _____

State : _____ Zip: _____ Country: _____

Phone : (___)_____-_____ Electronic Addr : _____

Option: ___ Both days, $55.00 by mail, $70.00 at the door
or: ___ First day, $35.00 by mail, $50.00 at the door
or: ___ Second day, $35.00 by mail, $50.00 at the door

Editorial

The MODUS Quarterly # 8, May 1987

Meeting

As you see from the front flyer in this issue there will be a MODUS meeting in San Francisco on June 15 and 16, Monday and Tuesday. The schedule of presentations is still open, so send in an abstract for your talk today to the MODUS address below. If you have questions about your presentation, then contact Stan Osborne at (415) 341-1768. The best papers presented will be printed in subsequent issues of MODUS Quarterly if the author permits.

Register via the MODUS address, P.O. Box 51778, Palo Alto, CA 94303. By mail, the registration fee is $55 or $35 per day. Registration at the door will be higher. Arrangements are being made to handle about 75 people at the meeting, so it might be important to get your registration in early lest the facilities overflow.

Prizes and Products

There are no new prizes awarded for either best suggestion or best article. This is due to a complete absence of either suggestions or commendations. Only you can improve this dreadful situation.

You may recall that issue # 7 had a page of new product announcements. We would like to devote a page or two of each issue to this sort of information. Please send us the relevant information about any new or updated product which deals with (or is written in) Modula-2. We do not print your advertisement itself so camera ready copy is superfluous.

Standards

You can see from this issue, some of the topics still being resolved by the British Standards Institution Working Group on Modula-2. They seem to be taking the job quite seriously and trying to learn from the problems perviously encountered in standardizing (they would say standardising) the Pascal language. Obviously, Pascal is rather similar to Modula-2. However, even with careful and extended effort, the result was imperfect.

This time, they get to try again with the benefit of the hindsight from the Pascal Standard. My understanding is that there will be three forms that the standard will take: A carefully phrased English definition of the semantics is intended to instruct people in the proper understanding of Modula-2. A second definition in the Vienna Definition Language is expected to permit proof of correct assertions about the language. Finally, a model implementation of Modula-2 is expected to make it easy for programmers to test out any peculiar constructs with a working (not necessarily efficient) system.

The general idea is that none of these three definitions overrides the others, but rather, if they ever disagree, then there is a bug in the standard which requires correction by the Working Group. Now, it may be that I misunderstand this intention, so read next issue's editorial to see what corrections have been made to my understanding. Meanwhile, I must say that if this in fact holds true, it may be the finest technical language standard ever developed.

rhk

The Atlantic Systems Guild Inc.

March 17, 1987

Dick Karpinski, Editor
MODUS Quarterly
6521 Raymond Street
Oakland, CA 94609

Dear Dick,

There is an old legal maxim that says, "silence gives consent." Publication in MODUS #6 of the papers of the British Standards Institution Working Group on Modula-2 is likely to be greeted with silence -- I am particularly concerned that any such silence not be taken as an indication of consent. I for one do not consent to what the group is trying to do.

Modula-2 is Niklaus Wirth's intellectual property. He gives the language gratis to those who wish to utilize it in its present form. He gives it, as well, to those who wish to incorporate its features (hopefully acknowledging their source) into some new language of their own. The only thing he does not give is the right to make changes to the language and then publish the result as Modula-2. This is exactly what the BSI group seems to be up to.

No properly construed standardization effort could consider such changes as adding underscores to identifiers or deleting the casting functions and reutilizing their syntax for conversions. Wirth has spoken clearly and eloquently on these matters. There was a standard before the BSI got involved. The only thing the group can hope to accomplish now is to destandardize the language by introducing a pseudo-standard in direct competition with Wirth's existing standard.

It is important that the changes advocated by Cornelius et al in the November 1986 MODUS not be debated on their merits. To do so would be to give explicit consent to the effort to "improve" Wirth's work. The group has no charter to undertake this effort. BSI can do real damage to Modula-2 by publishing a pseudo-standard. That's the practical consequence of the direction they've chosen. But the larger question is one of ethics. Wirth and Wirth alone can change the language; it's his. No body of law exists to defend Wirth's intellectual property. We, the members of his community have to defend it by rejecting BSI's attempt to tinker.

Sincerely,

Tom DeMarco

Tom DeMarco

353 W. 12th Street, New York, N.Y. 10014 (212) 620-4282
[illegible] London W2 1SU 01-262-3395

Response to DeMarco by R. Karpinski

I have been involved in standards works for over a decade so I can claim some understanding of what really happens in the process as practiced these years. The ANSI and IEEE language standard efforts always involve non-proprietary languages, so licensing questions do not arise. The buss field, on the other hand, does involve the use of patented devices and other licensed material. The IEEE project authorization process requires, in effect, that the vendor agrees to "reasonable" licensing fees as well as assurances that any changes the working group feels necessary will not be undermined by the vendor

Standards committees work without pay (except for individual members being paid by their own companies). In return for their work, the committees have wide latitude to do the best job that they can do, without regard to financial and marketing issues. To tie their hands in any way would question the basis on which standards are developed. The parent organizations, in practice, concern themselves only with issues of representation of all relevant interests among the voters, who are one level above the working group, and with fairness issues. Implicitly, they grant the working group complete autonomy in all questions of technical merit.

Most standards efforts are reactive. One outstanding exception is the pair of floating-point arithmetic IEEE standards, 754 & 854. These proactive standards establish a new (and far superior) floor for the operation of floating-point engines in computing systems. Efforts to eviscerate 854 were firmly rejected at the highest levels in the standards organization. The issues were for the working group to decide, not others.

In this light, reactive standards are expected to give high weight to existing practice. In particular, the base document from which the BSI draft of the Modula-2 standard is being developed is Wirth's then latest draft of the Modula-2 report. Indeed, the whole working group effort centers around the "Problems with the Modula-2 Report" which Barry Cornelius maintains (without necessarily agreeing with all the statements therein). These things suggest that the group does take seriously the origins and the spirit of Modula-2.

But why do we need a Modula-2 standard? I suggest that we need an official ISO Modula-2 Standard in order to assure potential users that there is a firm basis for their use of implementations of the language. In this case especially, with the bare, skeletal, nature of Modula-2, a usable standard library is a minimum requirement. Nobody, except possibly Wirth himself, considers the library given in "Programming in Modula-2" to be adequate. Several alternative libraries are in use, preventing the wide portability of Modula-2 applications.

Since there is disagreement in practice about the fine points of the language and the entire library, these things require resolution by a standards working group. Invoking the authority of the author will not be sufficient for the ISO. DeMarco (and Randy Bush) not-withstanding, current practice differs from PIM in too many serious ways not to be resolved by consideration of the merits of each case. For example, not even one vendor provides the dynamic linking capability which PIM-2 seems to require. Merits must be considered.

6 Nov 86

Dear Dr. Karpinsky,

I am addressing this letter to you, as it may be that MODUS is the proper vehicle for addressing some standards questions I would like to raise, in regard to Modula II. My programming involves significant amounts of real time data acquisition and real time graphics. Over the past several years, I have given much consideration to the appropriate language in which to begin a major redevelopment project. To date, I have written applications and portions of programs in various combinations of Pascal, FORTRAN, and assembly language. In my specific case, the applications are in the field of neurophysiology, but I am sure that my questions have a broad applicability. Based on my experience, I feel that Modula II represents the correct language to use for my proposed redevelopment project, which will involve rewriting (in a coherent manner) much of the software used in clinical and research neurophysiology.

Certain concepts seem to underly Modula II. One of them is the attempt to make it possible to write low level code that is relatively machine independent. Another is an attempt to make the language as small as possible, to allow its reasonable implementation in many environments. Certain other features appear to be designed specifically to allow the writing of interrupt handlers and device drivers directly without assembly code, and there seems to be an implied assertion that real time programming should be possible directly in Modula. As an important design concept, certain machine specific features, such as IO, have intentionally remained unspecified, and are to be provided by an implementor in the form of a library. The intention is to provide the tools to develop such facilities in whatever format is most appropriate for the given environment.

While most of the facilities for accomplishing these tasks are to be found within Modula, there are some that are lacking. In general, environment specific details, such as file IO, have been left for the libraries, as I mention. However, certain environment issues must be addressed, to provide a truly useful programming language. An example of such is the definition of the library procedures ALLOCATE and DEALLOCATE. While the implementation details remain the prerogative of the vendor, it would be very hard to imagine the utility of the language without some implementation of these procedures. Similarly, real time applications, particularly those (such as mine) that mark time in microseconds, do need a minimal core of interaction with the environment. Just as the availability of a standard definition of ALLOCATE and DEALLOCATE is crucial to the development of many applications, I believe the standardization of the approach to the needs of real time applications would be an important step.

I have identified three needs in such applications that I see as problematic, particularly in regard to the implementation of Modula compilers under the increasingly popular UNIX and related or derivative operating systems. Additionally, if there are other aspects of this problem that you feel nust be addressed, I am interested to hear about them. I have attempted, in the spirit of Modula II, to identify the least possible set of needs that could be implemented to support real time applications.

1) Handling of interrupts in real time: immediate access to data buffers.

In order to insure minimum latency between an interrupt and access to the data buffer where data is to be stored, such a buffer must be protected, or fixed in real hardware (not virtual) memory. I believe this is a real requirement, since there are at least two circumstances where virtual mapping would be disastrous: 1) If the OS decided to swap the array in and out of memory during a data acquisition period, at the very least valuable data would be lost, and 2) if a device (AD board) is capable of DMA, and is in the process of dumping data to a real physical location while a swap occurs, significant mayhem would result. It is necessary to know the true physical address of such a buffer, since the AD board often requires this information for DMA.

2) Interrupt handlers and interrupt service latency.

In order to program high speed interrupt handlers, I believe these must also be fixed in memory. In general, the virtual address must be known/determined at run time, in order that the appropriate trap vector be loaded with this address. However, since the task switching latency of an arbitrary harware configuration (read MMU and OS task handler) can never be guaranteed to be less than the requirements of a given application, it would be more generic and satisfactory (read necessary?) to provide a means of locking a handler into real physical memory. This problem was in fact alluded to, when it was decided not to provide any particular model of multitasking into Modula, but rather to provide the means to write multitasking systems **in** Modula.

Memory management considerations for implementing 1 and 2:

The critical portion of a handler is usually quite small, and would not degrade over-all system memory management, paricularly in today's hardware, where megabytes have become the rule. Likewise, data buffers would probably not cause problems, but since these can be large, provision can be made to protect these only briefly (i.e. require their residence in physical memory only at specified times), as almost by definition, the cases where this is necessary are those where sampling rates are very high, and therfore usually the buffer need only be locked for periods less than several seconds.

One "bare minimum" approach to this problem would be to provide a standard system call in the SYSTEM library. In its simplest form, this could consist of a procedure call:

```
Lock (Object, size_of_Object, pointer to Object);
UnLock (Object, pointer to Object);
```

where Object would be either an array or a handler, Lock would return the true location in the pointer, and UnLock could return a possibly different virtual location. It should be possible to create these objects dynamically, i.e. **not** require that they be known at compile time, as the data arrays may be very large, and only needed temporarily.

3) Transient processor locking (beyond priorities).

The other need that may also prove absolutely necessary, is control of the processor priority. While Modula does provide the capability to control the "priority" of a process, I think there is a subtly different need, namely the ability to completely lock the CPU temporarily, even at times to the extent of preventing any hardware interrupts, and certainly to the extent of preventing operating system interference or task switching. In other words, I forsee the situation where a process is declared "locally" (within a program) to be of the "highest" priority, yet is interrupted by the OS itself, running at a higher absolute priority or in a protected or "supervisor" mode.

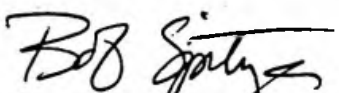
Again, possibly this need can be met by a pair of SYSTEM procedure calls, "LockCPU" and "ReleaseCPU":

```
LockCPU;
...time critical code here...
ReleaseCPU;
```

Thank you for your time in addressing this matter. I think that defining approaches to this problem would help provide a standard that vendors could adhere to, and thereby would make Modula a premier language for real time programming. I have noted several proposed "extensions" to Modula for various needs, including real time programming. I think my proposed solutions may meet the following desirable criteria: 1) They do not change the language. 2) Existing code probably need not be modified (the aforementioned procedures are only *necessary* in an environement where these facilities would have already been developed in a machine specific manner already, out of need). 3) No specific means of implementaion is specified. 4) I think these are the minimal set of operations needed: all other "extensions" could be developed using these.

The comments and proposals of the Modula community are greatly appreciated.

Sincerely,



A. Robert Spitzer MD

Date: 2 February 1987
To: R. Karpinski
From: Bill Nicholls
Subject: Open Letter from a Practicing Programmer

It was nice to receive my semiannual Modus Quarterly. Once again there was much of interest as well as some things which were difficult to make use of. But this issue contained a number of things which triggered some thoughts of my own. What I want to offer are some thoughts from the trenches about M2 and its possible use by others like myself.

A little background may help you understand my opinions. My training was in physics, my experience in computers almost totally OJT. I began programming 25 years ago in Fortran on an IBM 1620. Much of my early experience was assembler, and never a structured word was spoken. In the early 70's, I was introduced to structured programming indirectly by working with an Algol derivative called Totetran. My conversion to a structured approach was lengthy, difficult and clumsy since as usual, no formal education was provided.

But over a period of years I overcame many of the bad habits developed from writing Fortran, assembler and Basic. As a result, I understand better than most the issues around structured programming, having worked both sides of the fence so to speak. Today, structured programming is a given. Having achieved that with Pascal and C, we are now in the refinement stage with M2. So to me, the path to M2 and the justification is clear.

What I see both pleases me and worries me. The attention to standards and the effort to build a very workable standard is to be complimented. The language itself is very satisfactory with some minor quibbles. But what worries me is the broader issue of applying M2, even though some of these concerns are not restricted to M2 alone.

Software complexity has continued to increase over the years. Yet the support in the languages for dealing with increasing complexity has lagged badly, and the tools to deal with complexity even more. Most programmers simply laugh when you ask about code reuse. The difficulty of even reusing your own code never mind someone else's is enough to make most programmers avoid that chore.

M2 has taken a step in the direction of easing the reuse problem by separating the definition from the implementation. But it has in fact complicated the handling of source code by doubling the number of pieces needed, and

added considerable effort in the export/import requirements. Yet what results is still better than Pascal even if it is more difficult to handle in some cases.

Some of the extra steps can be handled with an appropriate programming environment, yet even that is less than sufficient. M2 is an excellent building block language yet there seems to be no simple automated ways to deal with the building blocks at a higher level. By this I mean that given a suitable library of M2 functions, I should be able to define the solution to a problem in some higher level terms and have an automated system builder put the blocks together, prompting me for decisions and new modules as needed.

By virtue of its clean definition and specification, M2 is probably the only language that this could be done in without requiring language extension. Such a system build facility would address my concerns about productivity by automatically taking care of most of the bookkeeping now required of the programmer. What I would hope to see would be a 'language' where the task could be defined in some set of terms similar to a 4GL and the system builder provide the detailed code as required, adding from the library as needed.

Such a system is not a 4GL since by definition it is extendable by the programmer by building new blocks and defining these to the system builder. This kind of capability would do much to reduce the overhead imposed by the extra M2 language requirements and provide a significant incentive for programmers to convert to M2

I think the incentive issue is a crucial one. Regardless of the M2 advantages over Pascal, C or (name your favorite language), the decision to use M2 as a primary tool is made very difficult by the barriers that must be overcome. First, there is the learning curve. You must learn the syntax and semantics, then the libraries and finally the issues of structure and efficiency. Then there is the lack of external library support, which is slowly being addressed. And finally there is the decision to translate, convert or rewrite all your own favorite routines.

Having looked at the barriers, you then evaluate the cost vs payoff. As it stands today, the cost exceeds the payoff significantly even though M2 is a better language from a number of points of view. The primary reason I draw this conclusion is the time spent in going up that learning curve is not repaid with an obvious productivity enhancement. I can see the benefits of the M2 approach, but they do not give me a clear bottom line advantage once I have made the investment.

Thus except for people choosing M2 when they begin programming and those with enough faith in the language to overcome the barrier, current

programmers will mostly just continue using their current language. This is unfortunate for those of us who would like to see better tools for and better products from programming. As good as M2 is, it fails to provide that productivity leap that will push most of us over the startup barrier.

I hope this has provided some food for thought for those people involved in language specification and compiler development.

Sincerely,

Bill Nicholls

BSI Modula-2 Working Group

Second Open Meeting
BSI Conference Centre
July 24th 1986

Coroutines and Processes

Roger Henry

Department of Computer Science
University of Nottingham
Nottingham NG7 2RD UK

Nottingham (0602 or +44 602) 506101 ext 2855

JANET: rbh@cs.nott.ac.uk

ABSTRACT

There are several problems with the definition of the coroutine mechanism in Modula-2. The semantics seem to be poorly understood and there is even doubt as to the status of coroutines within the language. Discussion of these issues by the BSI Working Group has recently started and a specialist subgroup has been set up. It has been agreed that a coroutine mechanism is to form part of the standard for the language and that a higher-level model for processes is to be included in the library. A distinction is made between the explicit scheduling of coroutines and the implicit scheduling of processes.

This paper reviews the original definition of coroutines and then describes an alternative mechanism as proposed by the subgroup. The old mechanism may be implemented in terms of the new (and the new in terms of the old) but is arguably simpler to comprehend and to use. Consideration is also given to the problems of processor priorities and monitors. The paper includes the results of discussions to date on the requirements for a library module based on the implicit scheduling of processes.

1. Introduction

Does Modula-2 provide facilities for multiprogramming?
If so, what are these facilities?

It is hard to give a straight answer to these questions given the current definitions and descriptions of the language. It is easier instead to answer a different pair of questions:

Do implementations of Modula-2 provide facilities for multiprogramming?
If so, what are these facilities?

The reason for this state of affairs is that there are no syntactic aspects of the language related specifically to multiprogramming†. Neither are there any standard procedures providing for concurrency. Instead, an implementation may provide access to low-level mechanisms *via* the pseudo-module SYSTEM and/or may

† With the possible exception of the optional priority specification in module headings.

provide higher-level facilities through a library module. This is consistent with the view of Modula-2 as a systems implementation language consequently offering different features on different systems. After all, it was designed to allow the actual construction of process schedulers such as the one built in to the predecessor language Modula.

However, the original reports and compilers emanating from Zurich, and the text *Programming in Modula-2* [1] all provide a model for a basic coroutine mechanism in SYSTEM. This can be used to implement a higher-level process abstraction on single processor systems. While this model has been criticised on the grounds of obscurity and lack of efficiency, it has been closely followed in many other implementations. However, successive versions of the report have progressively weakened its status. For example, the type SYSTEM.PROCESS was dropped and SYSTEM.ADDRESS used in its place. Originally, NEWPROCESS and TRANSFER were described as part of SYSTEM, then it was said that they were *normally* to be provided, and with the advent of the Lilith single-pass compiler it is stated as a *change* that they are not required [2].

The view of the BSI Working Group is that a substantial and important range of applications of Modula-2 depend upon the availability of a coroutine mechanism. The standard will therefore deal with the provision of coroutines in the language. Precisely the way in which this shall be done is yet to be decided but a specialist subgroup has been set up to make recommendations. I am the convener of this subgroup and other members are currently Don Ward (GEC Electrical Projects), Derek Andrews (Leicester University) and Willy Steiger (Logitech).

The essential nature of coroutines is that, within a set of such cooperating routines, only one routine is executing at a time and that transfer of control occurs explicitly to a nominated coroutine. An extension in Modula-2 is that external events such as interrupts may be preprogrammed to cause an asynchronous transfer. The term *coroutine* is to be used for this level, references to *process* being reserved for a model in which scheduling is implicit. Processes must be regarded as executing in parallel since in general there may be arbitrary interleavings of execution. While it is assumed that one important use of coroutines will be to implement processes on a single processor, it must not be forgotten that coroutines are an important tool in their own right and that a set of coroutines may be used by the program of a process in order to implement some algorithm. In the spirit of Modula-2, a minimalist approach is to be taken, defining only a neccessary and sufficient set of primitive types and operations within the language and leaving higher level facilities for library modules.

2. The original coroutine model

2.1. Synchronous transfers

In the beginning, the following type and procedures could be imported from SYSTEM†:

```
TYPE
    PROCESS;              (* coroutine state *)

PROCEDURE NEWPROCESS      (* dynamically create a coroutine *)
       (P: PROC;          (* code of coroutine *)
        A: ADDRESS;       (* base workspace address *)
        n: CARDINAL       (* workspace size in storage units *)
    VAR p: PROCESS        (* initial state of new coroutine *)
        );

PROCEDURE TRANSFER        (* synchronous coroutine transfer *)
   (VAR p1,               (* saved state of calling coroutine *)
        p2: PROCESS       (* state of destination coroutine *)
        );
```

† the type PROCESS is used here as was done originally by Prof. Wirth. Recognizing that this was misleading, later versions of the report replace PROCESS with ADDRESS. The Working Group have resolved to use COROUTINE in the standard.

It is assumed that the code making the first call of NEWPROCESS is being executed by a *main* coroutine. The initialization parts of all modules declared at level 0 form the program of this process and its workspace is set up by the implementation.

The state of a dynamically created coroutine is such that, on the initial transfer, execution will begin from the start of the parameterless level 0 procedure *P* forming the code of the coroutine. The entire program terminates if an attempt is made to return (implicitly or explicitly) from such an activation of *P*. On transfer back to a coroutine, the effect is for there to be a return from the call of TRANSFER in the destination routine. If it is known that there will never be an attempt to transfer back to a dynamically created coroutine, then the workspace may be reused and the coroutine has effectively terminated.

The workspace is used to hold coroutine description information needed by the implementation and for the coroutine stack. Hence local variables and procedure value parameters will be allocated in the workspace and become per-coroutine variables. There is only one instance of global (level 0) variables. Distinct workspace must be provided for active coroutines, but they may share the same code.

Note that much of this description is not given in the report and this has been supplemented by inferences drawn from examples in [1] and from experience with actual implementations. The Working Group takes the view that it shall be an exception for a dynamically created coroutine to attempt to return from the outer procedure forming its code. In this respect, the main coroutine differs in that it has been decided that returning from the main module body results in normal termination of the program.

2.2. A coroutine solution to the 8 Queen's problem

To illustrate the use of coroutines within a single process, here is a solution to the 8 Queen's problem.

```
MODULE Queens;

FROM Board IMPORT
    Safe,       (* tests if proposed placement is safe *)
    Occupy,     (* occupies a given place on the board *)
    Vacate,     (* vacates a given place on the board *)
    Display;    (* displays board *)

FROM SYSTEM IMPORT
    PROCESS,
    NEWPROCESS,
    TRANSFER,
    ADR,
    SIZE;

TYPE
    Arow = [1..8];
    Acol = [1..8];

CONST
    wkspsize = 200;     (* informed guess *)

VAR
    colData: ARRAY Acol OF
        RECORD
            cr: PROCESS;
            wksp: ARRAY [1..wkspsize] OF WORD;
        END;
    initCol: Acol;
    main: PROCESS;
    done: BOOLEAN;
```

```
PROCEDURE Placer;          (* coroutine code *)
    VAR
        myCol: Acol;
        myRow: Arow;
BEGIN
    myCol := initCol; TRANSFER(colData[myCol].cr, main);
    LOOP
        FOR myRow := 1 TO 8 DO
            IF Safe(myCol, myRow) THEN
                Occupy(myCol, myRow);
                IF myCol < 8 THEN
                    TRANSFER(colData[myCol].cr, colData[myCol+1].cr);
                ELSE
                    done := TRUE;
                    TRANSFER(colData[myCol].cr, main)
                END;
                Vacate(myCol, myRow);
            END
        END;
        (* none of the rows are safe in this column *)
        IF myCol > 1 THEN
            TRANSFER(colData[myCol].cr, colData[myCol-1].cr)
        ELSE
            done := FALSE;
            TRANSFER(colData[myCol].cr, main)
        END
    END
END Placer;

PROCEDURE Init;
BEGIN
    FOR initCol := 1 TO 8 DO
        WITH colData[initCol] DO
            NEWPROCESS(Placer, ADR(wksp), SIZE(wksp), cr);
            TRANSFER(main, cr)
        END
    END
END Init;

BEGIN
    Init;
    TRANSFER(main, colData[1].cr);
    WHILE done DO
        Display;
        TRANSFER(main, colData[8].cr);
    END;
END Queens.
```

2.3. Comments

This example illustrates the sharing of global variables such as *initCol* and *done*, the use of per-coroutine variables such as *myCol* and *myRow*, and the sharing of the coroutine code *Placer*.

Notice how correct use of TRANSFER depends upon the assumption that the PROCESS values will change during coroutine execution and must therefore be kept up to date *via* the first parameter. This usage is also relied on to obtain the state of the main process. Using an out of date PROCESS variable as the

second parameter to TRANSFER is equally dangerous and incorrect as dereferencing a dangling pointer.

Consider the following sequence of TRANSFERS between two coroutines:

```
Coroutine 1                    Coroutine 2

TRANSFER(A, B);
                        C := A;
                        TRANSFER(B, A);

TRANSFER(A, B);
                        TRANSFER(B, C);
```

What is the effect of the second TRANSFER in coroutine 2?

Does it return from the first TRANSFER in coroutine 1? The second? Or is it undefined? Interpreting PROCESS values as coroutine states, one might expect the first of these but that state cannot be recovered and so the likelihood is that the program will crash. However some implementors have chosen a fixed ADDRESS representation of coroutines and then the effect would be for a safe return from the second TRANSFER in coroutine 1.

What is the effect of TRANSFER(x, x) ?

This is not a null operation. The report explicitly states that the assignment to the first parameter occurs after identification of the destination process given by the second parameter. Thus two coroutines can arrange to swap execution by always giving the same PROCESS variable for both actual parameters. In general there is no need for a PROCESS variable to be associated with a unique coroutine. At different times it can correspond to the states of different coroutines. This change of reference is necessary for implementing the implicit scheduling of processes, but coming on top of the change in coroutine states the original coroutine model is far from being simple and safe to use.

2.4. Asynchronous transfers

Wirth has shown us how an interrupt handler can be viewed as a cyclic activity synchronizing with an externally generated event. In the model of interrupt handling provided in the original PDP11 implementation, synchronization is achieved by the interrupt handling coroutine making a call of

```
PROCEDURE IOTRANSFER     (* synchronous transfer out, asynchronous back *)
   (VAR p1,              (* saved state of calling coroutine *)
        p2: PROCESS;     (* way out - state of destination coroutine
                            way back - saved state of interrupted routine *)
        va: CARDINAL     (* interrupt vector *)
        );
```

This is equivalent to an immediate TRANSFER(p1, p2) and a subsequent involuntary TRANSFER(p2, p1) when the interrupt with vector address *va* is accepted. The old value of *p2* does not have to correspond to a previously interrupted coroutine since it may have been stored as a result of a TRANSFER. Similarly, an interrupted coroutine may be returned to by a TRANSFER.

The coroutine active at the time of the interrupt will certainly be in a different state from that saved as the old value of *p2* and, in general, it may be a different coroutine altogether if one or more TRANSFERs have intervened. It is therefore necessary in the implementation of process schedulers to use an auxilliary variable for p2 and copy values to and from the PROCESS variables associated with the appropriate coroutine. Note also that, while it is in order to copy PROCESS variables by assignment for later use, tests of equality on PROCESS values can not be used to determine the identity of coroutines because of the changing state.

Each and every interrupt must be preceded by a synchronizing call of IOTRANSFER.† Since, in general, interrupts must be enabled before IOTRANSFER is called, a way must be found to fend off the interrupt

† Without any intervening TRANSFERS to *p1*. Such an explicit TRANSFER could be programmed and would have the same effect as the occurrence of the interrupt, however it would only be safe to do this if the source of the interrupt had been disabled first.

until the first part of the IOTRANSFER has taken place. This is done by executing the critical code at a high processor priority as specified by the heading of the enclosing module. To guard against the possibility of spurious interrupts occurring when no IOTRANSFER has been issued, the safest policy is to enable interrupts before each IOTRANSFER and to disable them again immediately afterwards.

2.5. Buffered output to a serial terminal

This example is made relatively system independent by importing device specific constants and operations from another module *PrinterDevice*. It is assumed that the constant *devicePriority* is known in the enclosing module by import from *PrinterDevice*.

```
MODULE Printer[devicePriority];

FROM SYSTEM IMPORT
    NEWPROCESS,
    TRANSFER,
    IOTRANSFER,
    LISTEN,              (* momentarily lower processor priority *)
    PROCESS,
    WORD,
    ADR;

FROM PrinterDevice IMPORT
    Enable,              (* enable interrupts *)
    Disable,             (* disable interrupts *)
    Output,              (* output a given character directly *)
    vector;              (* interrupt vector address *)

EXPORT
    Print;

CONST
    N = 32;

VAR
    n: CARDINAL;         (* characters in buffer *)
    in,                  (* input pointer *)
    out: [1..N];         (* output pointer *)
    buff: ARRAY [1..N] OF CHAR;
    user,
    driver: PROCESS;
    wksp: ARRAY [1..100] OF WORD;
    waiting: BOOLEAN;    (* driver is waiting for a character *)

PROCEDURE Print(ch: CHAR);
BEGIN
    WHILE n = N DO LISTEN END;
    buff[in] := ch;
    in := in MOD N +1;
    INC(n);
    IF waiting THEN
        waiting := FALSE;
        TRANSFER(user, driver);
    END;
END Print;
```

```
PROCEDURE Handler;
BEGIN
    LOOP
        Enable;
        Output(buff[out]);
        out := out MOD N +1;
        DEC(n);
        IOTRANSFER(driver, user, vector);
        IF n = 0 THEN
            waiting := TRUE;
            Disable;
            TRANSFER(driver, user)
        END
    END
END Handler;

BEGIN
    n := 0; in := 1; out := 1; waiting := TRUE;
    NEWPROCESS(Handler, ADR(wksp), SIZE(wksp), driver);
END Printer;
```

2.6. Comments

It turns out that explicit scheduling is quite appropriate for the relatively tight coupling between user and device coroutines in this and similar examples. In general, however, there will be several user coroutines and these will be relatively loosely coupled within a framework of implicitly scheduled processes. A procedure such as *Print* must then not employ busy waiting when it finds its progress is blocked - in this case because of a full buffer. To solve this problem, both *Print*, executing the user coroutine, and *Handler*, executing the driver coroutine, must call on implicit scheduling operations in addition to the explicit scheduling operations already shown.

Modules which employ asynchronous coroutine transfers do introduce an issue which is also potentially present for interactions between quasi-parallel user processes. This is the issue of mutual exclusion in access to shared variables. Some of the possible interleavings of execution can lead to incorrect results if one coroutine uses a variable value while another alters the value of that variable. The solution employed in the *Printer* example is supported by the language and relies on raising the processor priority to exclude interrupts from the printer device while *Print* is being executed. (This is why LISTEN is called to momentarily lower the priority and admit any pending interrupts.) This action is implied by the priority specification in the module heading. Prof. Wirth calls such a module a *monitor* since only one coroutine can be in the state of executing a procedure of the module at a time except during calls of LISTEN, TRANSFER, and IOTRANSFER. Essentially what this is doing is limiting asynchronous transfers during execution of *Print* to the one point where LISTEN is called. Some of the implications of this will be taken up again after the discussion of implicit scheduling. However there are several unresolved issues to do with module priorities which are not specifically raised in this paper.

3. The proposed new coroutine model

3.1. Description

The Working Group have received many useful comments on the original coroutine model, especially from Dave Budgen (University of Stirling), George Mohay (Queensland Institute of Technology) and Pat Terry (Rhodes University). Much was also revealed by attempts to describe the model to some members of the Group who had yet to come to grips with Modula-2 coroutines! Taking account of this input, the specialist subgroup have devised a new model which is to be recommended for adoption in the standard. We feel able to do this because of the uncertain status of the original model, because of its relative complexity, and because the old model may be implemented in terms of the new thus imposing minimal changes on existing

code.

The most fundamental change is to replace the changing state of PROCESS values with fixed COROUTINE values associated with particular coroutines throughout their existence. This essentially introduces coroutine identifiers. A new function SELF is provided to allow the identity of the main process to be stored and for the occasions when a procedure needs to discover the identity of the executing coroutine. There is now no need for a parameter to TRANSFER and IOTRANSFER to save the state of the executing coroutine.

As a separate change, the operation of associating an interrupt source with the executing coroutine is separated from IOTRANSFER and is now performed by the new procedure ATTACH. Since in most if not all applications a handler will always use the same interrupt vector, this change will allow the vector to be set up once and for all and so will gain an important improvement in efficiency where speed is critically important.

Finally, the old second parameter of TRANSFER is split into two parameters. The first gives the value of the destination coroutine identity, the second is a VAR parameter which is set on return to the identity of the interrupted coroutine. In some applications the same variable will be chosen for each parameter but there are others in which the separation will save extra copying. In any case, the separation is claimed to be conceptually easier.

To overcome a potential problem with the original implementation of NEWPROCESS, NEWCOROUTINE is defined to set up the initial processor priority for execution of the new coroutine to be the same as that of the caller. In cases where NEWCOROUTINE is called indirectly through a higher level initialization routine, the call of NEWCOROUTINE should not therefore be enclosed in a module of specific priority. An alternative to this might yet be adopted and that is to have an (optional) extra parameter to NEWCOROUTINE to allow specification of the priority.

Here is the new model in the form of a definition module:

```
DEFINITION MODULE NewKernel;

FROM SYSTEM IMPORT
    ADDRESS;

EXPORT QUALIFIED
    COROUTINE,
    NEWCOROUTINE,
    TRANSFER,
    IOTRANSFER,
    ATTACH,
    SELF;

TYPE
    COROUTINE;                  (* coroutine identity *)

PROCEDURE NEWCOROUTINE          (* dynamically create a coroutine *)
        (Body: PROC;            (* code of coroutine *)
         WorkSpace: ADDRESS;    (* base workspace address *)
         WsSize: CARDINAL;      (* workspace size in storage units *)
    VAR cr: COROUTINE           (* identity of new coroutine *)
        );

PROCEDURE TRANSFER              (* synchronous coroutine transfer *)
        (To: COROUTINE          (* identity of destination coroutine *)
        );

PROCEDURE ATTACH                (* associate interrupt source with
                                   current coroutine *)
```

```
        (Vector: CARDINAL          (* interrupt vector *)
        );

PROCEDURE IOTRANSFER               (* Synchronous out, asynchronous back *)
        (To: COROUTINE;            (* identity of destination coroutine *)
    VAR From: COROUTINE            (* identity of interrupted coroutine *)
        );

PROCEDURE SELF                     (* deliver identity of current coroutine *)
        (): COROUTINE;

END NewKernel.
```

This model is thought to be safer since it is now not possible to attempt to transfer to a coroutine in an out of date state.

The new model has been implemented in terms of the old (and the old in terms of the new) by Don Ward. Synchronous transfers have been tested by him on the Logitech Modula-2 system for the DEC VAX running VMS. I have also tested these implementations using both synchronous and asynchronous transfers on a DEC PDP-11. The code is included as an appendix to this paper so that others may use it for comparison. Direct implementation for the PDP-11 is to be undertaken shortly.

It has not been finally decided from where the new types and procedures should be exported. There are four choices:

a) Keep them in SYSTEM. This is consistent with the low-level nature of the coroutine model but issues to do with the potential standardization of SYSTEM have yet to be resolved.

b) Export them from a system module other than SYSTEM. The compiler would not look for a compiled definition module and could generate efficient in-line code.

c) Export them from a separate (library) module.

d) Turn them into standard procedures and types which therefore do not need to be imported.

The working decision is to opt for b) and require a system module named COROUTINES. This should probably not be included as a standard identifier since that would imply that inner modules could import from COROUTINES without this being made apparent in enclosing modules. (However at least one member of the Working Group thought that for this reason SYSTEM should be made a standard identifier!)

3.2. Examples

The effect on the code of the 8 Queen's program is simply to do away with the first parameter in all calls of TRANSFER, and for a call of SELF to initialize *main* in the *Init* procedure:

```
MODULE Queens;

FROM Board IMPORT
    Safe,       (* tests if proposed placement is safe *)
    Occupy,     (* occupies a given place on the board *)
    Vacate,     (* vacates a given place on the board *)
    Display;    (* displays board *)

FROM NewKernel IMPORT
    COROUTINE,
    NEWCOROUTINE,
    TRANSFER,
    SELF;

FROM SYSTEM IMPORT
    ADR,
    SIZE;
```

```
TYPE
    Arow = [1..8];
    Acol = [1..8];

CONST
    wkspsize = 200;      (* informed guess *)

VAR
    colData: ARRAY Acol OF
        RECORD
            cr: COROUTINE;
            wksp: ARRAY [1..wkspsize] OF WORD;
        END;
    initCol: Acol;
    main: COROUTINE;
    done: BOOLEAN;

PROCEDURE Placer;        (* coroutine code *)
    VAR
        myCol: Acol;
        myRow: Arow;
BEGIN
    myCol := initCol; TRANSFER(main);
    LOOP
        FOR myRow := 1 TO 8 DO
            IF Safe(myCol, myRow) THEN
                Occupy(myCol, myRow);
                IF myCol < 8 THEN
                    TRANSFER(colData[myCol+1].cr);
                ELSE
                    done := TRUE;
                    TRANSFER(main)
                END;
                Vacate(myCol, myRow);
            END
        END;
        (* none of the rows are safe in this column *)
        IF myCol > 1 THEN
            TRANSFER(colData[myCol-1].cr)
        ELSE
            done := FALSE;
            TRANSFER(main)
        END
    END
END Placer;

PROCEDURE Init;
BEGIN
    main := SELF();
    FOR initCol := 1 TO 8 DO
        WITH colData[initCol] DO
        NEWCOROUTINE(Placer, ADR(wksp), SIZE(wksp), cr);
            TRANSFER(cr)
```

```
        END
    END
END Init;

BEGIN
    Init;
    TRANSFER(colData[1].cr);
    WHILE done DO
        Display;
    TRANSFER(colData[8].cr);
    END;
END Queens.
```

For the Printer example, the same variable is used for both parameters to IOTRANSFER since the driver always returns to the coroutine which it interrupted. The value of this variable will, in general, be changed by the call since a different coroutine may be running when the interrupt occurs.

```
MODULE Printer[devicePriority];

FROM NewKernel IMPORT
    COROUTINE,
    NEWCOROUTINE,
    TRANSFER,
    ATTACH,
    IOTRANSFER,
    SELF;

FROM SYSTEM IMPORT
    LISTEN,              (* momentarily lower processor priority *)
    WORD,
    ADR;

FROM PrinterDevice IMPORT
    Enable,              (* enable interrupts *)
    Disable,             (* disable interrupts *)
    Output,              (* output a given character directly *)
    vector;              (* interrupt vector address *)

EXPORT
    Print;

CONST
    N = 32;

VAR
    n: CARDINAL;         (* characters in buffer *)
    in,                  (* input pointer *)
    out: [1..N];         (* output pointer *)
    buff: ARRAY [1..N] OF CHAR;
    user,
    driver: COROUTINE;
    wksp: ARRAY [1..100] OF WORD;
    waiting: BOOLEAN;    (* driver is waiting for a character *)

PROCEDURE Print(ch: CHAR);
BEGIN
```

```
    WHILE n = N DO LISTEN END;
    buff[in] := ch;
    in := in MOD N +1;
    INC(n);
    IF waiting THEN
        waiting := FALSE;
        user := SELF();
        TRANSFER(driver);
    END;
END Print;

PROCEDURE Handler;
BEGIN
    ATTACH(vector);
    LOOP
        Enable;
        Output(buff[out]);
        out := out MOD N +1;
        DEC(n);
        IOTRANSFER(user, user);
        IF n = 0 THEN
            waiting := TRUE;
            Disable;
            TRANSFER(user)
        END
    END
END Handler;

BEGIN
    n := 0; in := 1; out := 1; waiting := TRUE;
    NEWCOROUTINE(Handler, ADR(wksp), SIZE(wksp), driver);
END Printer;
```

It is recognized that different systems may need to provide alternative versions of ATTACH to allow appropriate identification of the source of the interrupt. The portability of the *Printer* module could be enhanced even further by importing an *Attach* procedure from *PrinterDevice* instead of *vector*.

4. Implicit scheduling and processes

It is thought to be desirable to specify a library module providing for implicitly scheduled coroutines or *processes*. Discussion of the detailed interface and semantics has not yet proceeded very far but several requirements have emerged. Some aspects of the following description have deliberately been left vague at this stage.

a) There is to be a procedure *StartProcess* which creates a new process and makes it ready for execution. This means that it is a candidate for execution when scheduling takes place.

b) Return from the procedure forming the code of the process is to have the effect of terminating only that process. This is achieved by having the nominated procedure called indirectly by hidden coroutine code.

c) The effect of a return from this procedure may also be obtained by a call to *StopMe*.

d) For workspace, *StartProcess* is only to be given the increment on size needed over and above the minimum for the implementation. The actual space will be allocated by StartProcess and deallocated on process termination.

e) The procedure given by the user as the program of the process is to be of a type with one value parameter. The actual value to be used in the call of the procedure is to be given as a parameter to

StartProcess. The type of the parameter has not yet been decided - suggestions are CARDINAL or ADDRESS.

f) A procedure *Reschedule* will cause the current process to give up execution but remain ready.

g) A procedure *SuspendMe* will cause the current process to give up execution and become unready.

h) There shall be process identities discoverable from a call of *Me*. (I also suggest that there be a distinguishable *nilProcess* value for *Process* variables.)

i) A procedure *MakeReady* will be provided to make an identified unready process ready for scheduling.

These requirements are thought to provide a reasonable level of functionality for implicit scheduling. For example, they allow a solution to the busy waiting problem in the code of the *Printer* example. Here, *blocked* is a *Process* variable initialized to *nilProcess*.

```
PROCEDURE Print(ch: CHAR);
BEGIN
    IF n = N THEN
        blocked := Me();
        SuspendMe;
    END;
    buff[in] := ch;
    in := in MOD N +1;
    INC(n);
    IF waiting THEN
        waiting := FALSE;
        user := SELF();
        TRANSFER(driver);
    END;
END Print;

PROCEDURE Handler;
BEGIN
    ATTACH(vector);
    LOOP
        Enable;
        Output(buff[out]);
        out := out MOD N +1;
        DEC(n);
        IF blocked <> nilProcess THEN
            MakeReady(blocked)
        END;
        IOTRANSFER(user, user);
        IF n = 0 THEN
            waiting := TRUE;
            Disable;
            TRANSFER(user)
        END
    END
END Handler;
```

Such a scheme is also potentially applicable when running processes under an operating system which does not allow access to device interrupts but does offer routines to be called on the completion of asynchronous transfers. Provided that the implementation allows Modula-2 procedures to be specified as completion routines, they may call *MakeReady* to allow a blocked process to continue.

Higher levels of functionality may be built on top of this model. Thus other modules can implement semaphore operations or allow the use of *CoBegin*, *CoStart*, and *CoEnd* to start and wait for several subsidiary

processes to finish. The aim of providing any or all of these library modules is not to restrict the user but to offer him or her something useful to get started. The proposed COROUTINE mechanism is powerful enough to allow other schemes including one in which interrupt routines are more fully integrated into the process model [3].

No assumptions have been made about the actual scheduling algorithm to be used except that it must be fair. There may or may not be preemptive timeouts. The topic of mutual exclusion must therefore be raised again. The basic choice is between using a procedural mechanism such as claiming and releasing semaphores, or to use module priorities to create simple monitors. A limitation of monitors based on processor priorities is that monitor procedures risk releasing exclusive access by calling the procedures of another such monitor. This would happen if the procedure of the second monitor caused a call to LISTEN to be made, for example. A generalization to the language is under consideration which would allow user provided code to be executed on entry and exit to monitor procedures instead of the usual in-line code to raise and restore processor priority.

A further requirement of processes is that within each process a set of coroutines may be employed - for example in the way they are for the 8 Queen's example. There are two possible ways to provide for this. First the user could conceivably access the low-level COROUTINE mechanism directly. If so, the process module would need to be written on the assumption that COROUTINE identities were not fixed within a process. This gets us back full circle since the COROUTINE variable in the process descriptor would have to be updated on every implicit transfer.

The second approach would be to provide a higher-level coroutine library module the implementation of which knows about processes and updates the COROUTINE identity stored for the current process on each explicit transfer. An advantage here is that the implementation could guard against attempts to transfer between coroutines belonging to different processes. A high-level coroutine module could also allow for parameters and storage allocation.

Anyone out there asking for coroutines to start their own processes can either keep quiet or join the subgroup!

5. References

1. N. Wirth, *Programming in Modula-2*, Springer-Verlag, 1st edition: 1982, 2nd edition: 1983, 3rd edition: 1985.
2. N. Wirth, *A Single-pass Modula-2 Compiler for Lilith*, 1.5.84/rev. 15.11.85.
3. R. Henry, *Modula-2 Processes - Problems and Suggestions*, MODUS Quarterly, Issue 4, November 1985.

6. Appendix - new and old style kernel implementations

```
IMPLEMENTATION MODULE NewKernel;

IMPORT
        SYSTEM,
    Storage;

(* This implementation uses dynamic storage
   for clarity of code and therefore generates
   "garbage". An alternative would be to take
   the necessary storage from the given
   workspace *)

TYPE
    COROUTINE = POINTER TO
        RECORD
                        Worksp: SYSTEM.PROCESS;
                        IOVector: CARDINAL;
```

```
                END;

CONST
        NILVector = 0; (* System Dependent *)

VAR
        CurrentRoutine: COROUTINE;

PROCEDURE NEWCOROUTINE(
        Body: PROC;
        WorkSpace: SYSTEM.ADDRESS;
        WsSize: CARDINAL;
    VAR cr: COROUTINE
        );
BEGIN
        Storage.ALLOCATE(Me, SYSTEM.SIZE(cr^));
        cr^.IOVector := NILVector;
        SYSTEM.NEWPROCESS(Body, WorkSpace, WsSize, cr^.Worksp);
END NEWCOROUTINE;

PROCEDURE SELF(
        ): COROUTINE;
BEGIN
        RETURN CurrentRoutine
END SELF;

(***************)
MODULE Safe [7];
IMPORT
        SYSTEM, COROUTINE, CurrentRoutine;
EXPORT
        TRANSFER, IOTRANSFER;

        PROCEDURE TRANSFER(
                To: COROUTINE
                );
        VAR
                MySelf: COROUTINE;
        BEGIN
                MySelf := CurrentRoutine;
                CurrentRoutine := To;
                SYSTEM.TRANSFER(MySelf^.Worksp, To^.Worksp);
        END TRANSFER;

        PROCEDURE IOTRANSFER(
        To: COROUTINE;
        VAR From: COROUTINE
                );
        VAR
                MySelf: COROUTINE;
                temp: SYSTEM.PROCESS;
        BEGIN
                MySelf := CurrentRoutine;
        temp := To^.Worksp;
```

```
        CurrentRoutine := To;
                SYSTEM.IOTRANSFER(MySelf^.Worksp, temp, MySelf^.IOVector);
                CurrentRoutine^.Worksp := temp;
        From := CurrentRoutine;
                CurrentRoutine := MySelf;
        END IOTRANSFER;

END Safe;
(*******)

PROCEDURE ATTACH(
        Vector: CARDINAL
        );
BEGIN
        CurrentRoutine^.IOVector := Vector;
END ATTACH;

BEGIN
        Storage.ALLOCATE(CurrentRoutine, SYSTEM.SIZE(CurrentRoutine^));
        CurrentRoutine^.IOVector := NILVector;
END NewKernel.
```

```
DEFINITION MODULE OldKernel;

FROM SYSTEM IMPORT
        ADDRESS;

EXPORT QUALIFIED
        PROCESS,
        NEWPROCESS,
        TRANSFER,
        IOTRANSFER;

TYPE
        PROCESS;

PROCEDURE NEWPROCESS(
        Body: PROC;
        WorkSpace: ADDRESS;
        WsSize: CARDINAL;
    VAR pr: PROCESS
        );

PROCEDURE TRANSFER(
    VAR From,
        To: PROCESS
        );

PROCEDURE IOTRANSFER(
    VAR From,
        ToFrom: PROCESS;
        Vector: CARDINAL
        );

END OldKernel.
```

```
IMPLEMENTATION MODULE OldKernel;

IMPORT
        NewKernel;
FROM SYSTEM IMPORT
        ADDRESS;

TYPE
        PROCESS = NewKernel.COROUTINE;

PROCEDURE NEWPROCESS(
        Body: PROC;
        WorkSpace: ADDRESS;
        WsSize: CARDINAL;
    VAR pr: PROCESS
        );
BEGIN
        NewKernel.NEWCOROUTINE(Body, WorkSpace, WsSize, pr);
END NEWPROCESS;

(***************)
MODULE Safe [7];

IMPORT NewKernel;

EXPORT TRANSFER, IOTRANSFER;

PROCEDURE TRANSFER(
    VAR From,
        To: PROCESS
        );
VAR
        temp: PROCESS;
BEGIN
        temp := To;
        From := NewKernel.SELF();
        NewKernel.TRANSFER(temp);
END TRANSFER;

PROCEDURE IOTRANSFER(
    VAR From,
        ToFrom: PROCESS;
        Vector: CARDINAL);
BEGIN
        NewKernel.ATTACH(Vector);
        From := NewKernel.SELF();
    NewKernel.IOTRANSFER(ToFrom, ToFrom);
END IOTRANSFER;

END SAFE;
(*******)

END OldKernel.
```

To: Members of the BSI Modula-2 Working Group

From: Barry Cornelius
Department of Computer Science
University of Durham
Durham DH1 3LE England

Durham (0385 or +44 385) 64971 extension 792

Barry_Cornelius@uk.ac.durham.mts
Barry_Cornelius%mts.durham.ac.uk@UCL-CS.ARPA
Barry_Cornelius%DURHAM.MAILNET@MIT-MULTICS.ARPA
bjc@uk.ac.nott.cs
bjc%cs.nott.ac.uk@UCL-CS.ARPA

Ref: M2WG-N110

Title: Another look at the FOR statement

Version: 1

Date: 21st August 1986

1. Introduction

We last discussed FOR-statements at the M2WG meeting held in Leicester on 20th May 1986. In particular, we discussed the three possible definitions of a FOR-statement given on pages 3 to 5 of Derek Andrew's paper "Some Problems with Modula-2" (M2WG-N96).

The majority of those present agreed to the second definition, namely:

```
temp1:= e1;
temp2:= e2;
i:= temp1;
WHILE i<=temp2 DO
   body;
   i:= next(i, e3)
END ;
i:= UNDEFINED;
```

(assuming that the value of e3 is greater than zero). This definition is different to that of ISO Pascal and at the meeting I did not support the decision to adopt this definition.

In this document I give reasons for my opposition and produce a definition for the Modula-2 FOR-statement that agrees with ISO Pascal.

2. The Difference Between The Above Definition And ISO Pascal

The above definition doesn't allow a FOR-statement like:

```
VAR i: [ 1..10 ];
FOR i:= 42 TO 27 DO
...
```

Although this doesn't look very useful, the ISO Pascal Standard regards this as legal. In fact, it goes out of its way to allow such FOR-statements. It says:

> The control-variable shall possess an ordinal-type, and the initial-value and final-value shall be of a type compatible with this type. The initial-value and final-value shall be assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

Why does it do this? Well, there are occasions when you want a FOR loop executed zero times and the initial value has a value which does not belong to the type of the control-variable. The next section gives some realistic examples.

3. Some Realistic Examples

In this section I give two examples of code which is legal Pascal but which would not be correct according to the definition proposed in Section 1.

3.1 Transposing A Matrix

One algorithm for transposing a matrix is:

```
for each row of the matrix
   perform interchanges on columns
                which are to the right of the leading diagonal
```

Note that on the last row of the matrix there are no columns to the right of the leading diagonal so we don't want any interchanges performed. However, I don't regard this as anything strange and would argue that it is reasonable to produce the code:

```
CONST  size=10;
...
TYPE  matrix=ARRAY [1..size, 1..size] OF real;
...
PROCEDURE transpose(VAR a:matrix);
   VAR  row, col:1..size;  oldvalueofarowcol:real;
BEGIN
   FOR row:= 1 TO size DO
      FOR col:= row + 1 TO size DO
      BEGIN
         oldvalueofarowcol:= a[row, col];
         a[row, col]:= a[col, row];
         a[col, row]:= oldvalueofarowcol
      END
END { transpose }
```

3.2 Reading A Line Of Characters

My colleague, Robin Stanaway, uses this example in his lectures:

```
CONST
  maxlinelength = 80 {for example};

TYPE
  linelengths = 0..maxlinelength;
  linepos = 1..maxlinelength;
  lines = RECORD
            contents: PACKED ARRAY [linepos] OF char;
            length: linelengths;
          END {lines};

PROCEDURE ReadLine (VAR line: lines);
  {Reads a line of text and stores it, up to the maximum
   permitted length. Any excess characters are skipped over}
  VAR
    pos: linepos;
BEGIN
  WITH line DO
    BEGIN
      length := 0;
      WHILE (length < maxlinelength) AND NOT eoln DO
        BEGIN
          length := length+1;
          read (contents[length]);
        END;
      (*  (length=maxlinelength)  OR  eoln  *)
      FOR pos := length+1 TO maxlinelength DO
        contents[pos] := ' ';
    END;
  readln;
END {ReadLine};
```

4. A First Attempt At A New Definition

I do not see any reason why the FOR-statement of Modula-2 should be any different from that of Pascal (other than the changes required by the introduction of the BY-part). I would therefore like to propose the following definition for the semantics of the FOR-statement:

```
temp1:= e1;
temp2:= e2;
WHILE temp1<=temp2 DO
   i:= temp1;
   body;
   temp1:= next(temp1, e3)
END ;
i:= UNDEFINED;
```

where temp1 and temp2 are variables whose type is the same as the base-type of the control-variable.

I believe that this is just as easy to understand as the definition given in Section 1. The only difference is that the control-variable only gets its new value if the body of the loop is to be executed.

5. Other Problems

However, there are problems both with the definition in Section 1 and with this new definition.

The definition given in Section 1 doesn't cope with:

```
VAR colNum: [ 1..80 ];
FOR colNum:= 1 TO 80 DO
...
```

since the definition's equivalent code will eventually attempt to assign the out-of-range value, 81, to the control-variable, colNum.

And, neither the definition given in Section 1 nor the one given in Section 4 will cope with:

```
VAR day: (sun, mon, tue, wed, thu, fri, sat);
FOR day:= sun TO sat BY 2
...
```

since the equivalent codes will attempt to evaluate "next(day, 2)" when day has the value sat.

6. A Second Attempt At A New Definition

If we want to solve these problems then it seems that the definition will necessarily become more complicated. The best I can come up with is:

```
temp1:= e1;
temp2:= e2;
LOOP
   IF temp1>temp2 THEN EXIT END;
   i:= temp1;
   body;
   FOR temp3:= 1 TO e3 DO
      IF temp1=MAX(t)
         THEN EXIT
         ELSE INC(temp1)
      END (* IF *)
   END (* FOR *)
END (* LOOP *) ;
i:= UNDEFINED;
```

where:

(i) t is the base-type of the control-variable i,
(ii) temp1 and temp2 are both of the type t,
(iii) temp3 is of type CARDINAL.

Automatic export of identifiers from the definition module

A H J Sale

Professor of Information Science — University of Tasmania
GPO Box 252C Hobart Tasmania Australia 7001

The BSI Modula-2 Working Group has suggested that Professor Wirth's proposal to discard the export list of a definition module be rejected (Cornelius, 1986). In their report to the Modula-2 Users' Association the Working Group state

'[WG084] M2WG has agreed to retain the original syntax and semantics, i.e., objects which are to be exported from a definition module have to be listed in an export list.'

This paper argues that this suggestion should not be adopted and that the 1984 revision to automatically export all identifiers declared in a definition module should be retained.

ANALYSIS

Several correspondents to the MODUS Quarterly (November 86) have pointed out that few if any definition modules export identifiers selectively. This provides a strong *prima facie* case for assuming that the export clause in a definition module is redundant. There is also a strong case on theoretical and consistency grounds for rejecting selective export. However a more careful analysis of the situation is required to ensure that some unusual but important facility is not being overlooked by deleting it. Accordingly this paper will systematically examine the arguments for and against selective export. In the ensuing discussion the word *visible* (and its derivatives) will be meant to refer to an item being readable in the text of the definition module, and the term *accessible* to refer to an ability to refer to the item by imported identifier in a using module.

Firstly, we should ask when an identifier *must* appear in a definition module. Assume that the valid reasons for an identifier appearing in a definition module are based on it it serving a syntactic purpose in either a calling module or the definition module. The following syntactic purposes can be identified:

- the identifier will be exported, or
- the identifier is required for a subsequent using occurrence in the definition module, or
- the identifier is required as a by-product of one of the other two requirements.

Secondly, would anyone wish to transfer the defining occurrence of an identifier from an implementation module to the corresponding definition module? The only plausible reason for such a practice would be that the writer of the module wishes to not publish the text of the implementation module *and* publish the structure or definition of some object which is part of the implementation. This practice should not be encouraged. The definition module should constitute the module writer's contract with the user, and extraneous material should not be included in it.

Selective export allows the module writer to include such defining occurrences in the definition module and yet not export them; the deletion of the export clause would remove this possibility. However this argument for its retention is weak and a similar purpose can be achieved, if absolutely necessary, by either publishing the text of the implementation

module or by including a suitable comment in the definition module. This point is crucial, because it will be shown that there is no other plausible use for selective export.

VALID DEFINING OCCURRENCES

Procedures

A definition module may only contain a procedure heading, not its body. While the heading may contain a parameter list, the parameter identifiers have no significance to the user of a module and are never exported. Their only significance, if any, is for checking against the (redundant) heading in the implementation module. The only components of a procedure which are exported are its identifier and the structure of its parameter list.

Since no bodies of procedures can occur in a definition module, and the definition module has no initialization section of its own, there can be no using occurrences of the procedure identifier (calls or activations). This means that a procedure can never be required as the consequence of something else—it can only be a valid component of a definition module if it is to be exported.

Variables

It has been argued in many places that modules should not export variables as they represent severe security risks to the integrity of the module's correctness. For example see Sale (1986b). However, if a variable does occur in a definition module, there can also be no using occurrences (references) to it, for precisely the same reason as for procedures. Again a variable can never be required as the consequence of something else—it can only be a valid component of a definition module if it is to be exported.

Constants

Now consider the case of constants declared in a CONST part. These may be intended for export only, or may be referenced in subrange type declarations in the definition module. The first case is relatively rare in practice but can occur as defining the maximum size of some resource, or in providing identifiers for common constant values such as the ISO character set. Other examples are given in Sale (1986a). Not exporting an identifier intended for export is senseless.

The second case is much more common, yet it still does not offer an argument for selective export, for if the constant defines one of the limits of a user-accessible subrange type, then the limit values are valid information for the user and may be useful in FOR statements (for example). In any case it would be pointless to try to hide them because the accessibility of the type enables their values to be retrieved by the MIN or MAX functions, or failing that by the appropriate type transfer function (inverse to ORD). The only facility not available to a module user who is given exported access to a subrange type but not its limits is the facility to use (syntactic) constant expressions involving the limits, thus prohibiting the declaration of derived constants or subrange types. Even this would be possible if the proposal to allow standard function calls (eg MIN, MAX) in constant expressions is implemented.

Types

Since no other kind of object requires selective export, any case for it must lie in the area of type declarations. One possibility is for an exported type to be defined in the TYPE part of a definition module, but for any named identifiers in its internal structure to be not exported. This is a sort of opaque export: the structure is visible but cannot be used. Let us examine the possibilities:

Type synonyms
Declarations of the kind

Identifier1 = Identifier2

are relatively rare in definition modules, but introduce no new issue. Since the type is simply given a synonymous name, the purpose of this is probably to export it, but in any case whether selective export is important or not depends on whether the type Identifier2 is accessible anyway.

Subrange types
If the host type is accessible to the user, then not exporting a visible subrange type is almost pointless. Type identity in variable parameter compatibility is the only case where similarly declared subrange types are distinguished.

Enumeration types
A new issue is introduced with enumeration types. Does it make sense to export an enumeration type identifier but none of the associated constant identifiers? Or to export only some of the associated constant identifiers? The first question is almost equivalent to exporting the identifier opaquely, and is discussed later. The second implies that there are values which the user can see but not access by identifier. Since all values of a visible enumeration type can be reconstructed anyway by type transfer (given that they are visible), this seems pointless. The *Modula-2 Report* (Wirth, 1982) and most Modula-2 compilers with explicit export resolve this issue by simply not allowing selective export of enumeration types: if the type identifier is exported, so are all the constant identifiers which may not themselves be explicitly exported.

Record types
Of all the structured types, only record types involve the defining occurrences of internal identifiers: the field identifiers. These identifiers are not in the scope of the definition module and correspondingly are not in the scope of an export clause. There is only one sensible approach to the export of field names and this is stated in the *Modula-2 Report*: they are exported associated with the type identifier.

Set types, array types and pointer types
These types involve no internal defining occurrences.

Procedural types
Procedural types involve no internal defining occurrences. In any case, the visibility of a procedural type declaration (regardless of export) allows a user to reconstruct compatible types and procedures, since compatibility is determined by structural rules, not type identity. There is no reason at all for not exporting a visible procedural type.

Opaque export
Many implementations of Modula-2 restrict opaque export to types which are subsequently declared to be pointer types. This restriction is a compiler convenience, as it permits simple implementations. However, even with it, implementation modules can be written which provide any desired type—all that is necessary is for the pointer type to have the desired type as its bound type. It should also be pointed out that there are compilation techniques for

entirely removing this restriction. A quality Modula-2 implementation could permit an opaquely exported type to be declared in the implementation module with any type.

SUMMARY

The valid inclusion of a variable or procedure in the definition module implies its export. (If this is not the case then the definition module is overspecified and the non-exported object should be transferred to the implementation module.) The selective export of constant identifiers has been shown to have no useful purpose. The only possible case for selective export arises with type declarations. This is focused on one issue: the export of a type identifier whose defining occurrence occurs in the definition module but the non-export of any internal identifiers in its structure. The user really wants an opaque export of a structured type and attempts to achieve this by having the structure visible but not exported. This is a bad response to a poor situation: the proper solution is to encourage Modula-2 processors to implement opaque export so that the details of any type in the implementation module can be opaquely exported. There are techniques for doing this. Alternatively the existing opaque export of a pointer type can be used to provide the desired facility. The conclusions are simple:

1 Deletion of the export clause and automatic export of all identifiers whose defining occurrence occurs in the definition module is an improvement in and a simplification of the language.

2 Why should *local modules* retain their own idiosyncratic structure? Should they not also have interface and implementation components syntactically parallel to separate modules?

REFERENCES

CORNELIUS, B. (1986). 'Significant changes to the Language Modula-2.' *MODUS Quarterly*, 6, pp8–14.

SALE, A H J. (1986a). *Modula-2 : Discipline & Design.* Addison-Wesley.

SALE, A H J. (1986b). 'Improving the quality of definition modules.' *MODUS Quarterly*, 6, pp27–29.

SALE, A H J. (1987). 'Optimization across module boundaries.' *Aust. Comp. Jnl.* (to be published 1987).

WIRTH, N. (1982). *Programming in Modula-2.* Springer-Verlag.

BSI Modula-2 Working Group

Standard Concurrent Programming Facilities

N 116 Issue 3

Don Ward

Systems Design Division
GEC Electrical Projects Ltd
Boughton Road
Rugby
Warwickshire CV21 1BU UK

Rugby (0788 or +44 788) 2144

ABSTRACT

This paper outlines the proposals made by the processes subgroup of the BSI Working group on Modula-2. It discusses some of the various options when considering what to standardise and what to leave out and makes recommendations for adoption in the standard.

CONTENTS

1. Introduction

This paper reports on the work done by the processes subgroup to date. The group consists of the following members of IST 5/13

Derek Andrews (Leicester university)
Roger Henry (Nottingham University)
Willy Steiger (Logitech)
Don Ward (GEC Electrical Projects)

Paul Manning of Leicester University has also attended our meetings. This document has greatly benefited from discusions within the group, and especially from suggestions made by Roger Henry.

Terminology

PIMn	Programming in Modula-2 Edition n [1]
Priority	Hardware Priority
Importance	Software Priority
Monitor	Monitor as defined by C.A.R. Hoare
Uninterruptable Module	Monitor as defined by N. Wirth in PIM
Coroutine	PROCESS as defined in PIM1 & PIM2
Process	Coroutine which is implicitly scheduled
M2WG	BSI Modula-2 Working Group
(Nxxx)	Working document no. xxx of M2WG

2. Scope of Standardisation

There are a number of viable alternatives open to the standardisation working group when considering what concurrent programming facilities to standardise and what to leave out.

Standardise nothing

> "No consensus on a general machine independent model for concurrency yet exists, or indeed can be expected to at this stage ... the language standard should not prescribe any such model, but should leave implementations to provide appropriate facilities via suitable library modules" .... N. Wirth as reported by Jim Welsh and Paul Bailes (The go-betweens' tale N105).

Standardise Priority only

> Recognise the truth of NW's comments but include at least a method of providing non-interruptability in a structured way within the language.

Standardise Coroutines

> Argue that coroutines are well understood, fairly portable (except workspace size) and should be included within standard Modula-2. The position of IOTRANSFER is less solid, but a number of implementations have managed to provide it. If IOTRANSFER is required, it is strongly recommended that priority is required too.

Standardise Processes

> Argue that 'raw coroutines' are difficult to use and that modules which provide some kind of implicit scheduling together with commonly used synchronisation facilities and process creation facilities would be useful. 'Rolling your own' should not be prohibited by making them part of the language - but neither should reinventing the wheel be encouraged.

Standardise a toolbox

> Take the wheel reinvention argument to it's logical limit and provide a rich set of concurrent programming facilities.

The position of the subgroup is to propose that coroutines and processes be standardised. We were encouraged by the feedback from the open meeting in July 1986 which, broadly speaking, endorsed this view.

3. Summary of the Proposal

We propose a definition of a required system module called COROUTINES for the explicit transfer of control between coroutines. This definition is implementable in terms of the old definition of coroutines (aka. processes) in PIM and vice versa.

We propose that there be a type PRIORITY with at least two values (priorities) - Interruptable and Uninterruptable. Implementations are free to add intermediate priority levels between these two mandatory ones. We propose PRIORITY should be defined in COROUTINES.

We propose two required separate modules to do implicit scheduling and to implement a general semaphore operation.

We do not require a module which has an explicit priority specification to be a monitor (in the Hoare sense) nor do we require anything else from the apparatus available to the concurrent programmer (who is of course free to provide it himself and to reimplement our required separate modules to his or her own taste).

4. Priority

We require two priority levels Interruptable and Uninterruptable. An implementation is free to augment this by inserting a (partially ordered) set of priorities between the two required values.

Interruptable < Implementation defined (partially ordered) set of priorities < Uninterruptable

If the *poset* is totally ordered we have the familiar priority levels known to fans of the PDP11. Note that it is still possible to make rules about whether a procedure running at one priority may call one which runs at a different priority even if the set of priorities is not totally ordered. One can define two priorities to be incomparable (if one is not a subset of the other) and define the priority which is a strict subset to be the lower of the two.

The effect of the two required priorities is as follows:

Uninterruptable
: No coroutine which has performed an IOTRANSFER will return from it while the priority is Uninterruptable.

Interruptable
: No coroutine is prohibited from returning from an IOTRANSFER operation.

The priority of the processor may be changed by the current coroutine entering or leaving a module with a priority explicitly specified in the heading (ie. calling or returning from a procedure defined within such a module or obeying the initialisation code of the module itself). The priority may also be changed by TRANSFERing to another coroutine. If a source of interrupts is excluded and a procedure is called (or a module is initialised) which changes the priority, the source of interrupts must remain excluded.

Note that if a procedure variable or a procedure parameter is used, the checking that the priority is not lowered must be deferred until run time.

If a module, which explicitly specifies a priority, is textually within another module, which also specifies a priority, then the priority of the inner module cannot be lower than or incomparable with the priority of the outer[1]

The priority does not have to be cpu priority wrt. physical devices: An implementation which never turned interrupts off (as far as the devices were concerned) but which stored them up and delivered them when they were no longer excluded by the current priority would be acceptable. This approach is likely if the implementation is not running on a bare machine.

If more than one interrupt is delivered from a particular source before it is accepted (by lowering the priority to allow a return from the IOTRANSFER), it is implementation dependant whether subsequent interrupts are ignored or recorded.

If the priority allows an interrupt and one occurs from an attached device, there will be an exception unless a coroutine, attached to the source of interrupts, has previously suspended itself using IOTRANSFER. It is implementation dependent whether interrupts from devices to which no coroutine is attached give rise to exceptions or not.

4.1. The type PRIORITY

We define a type PRIORITY with (at least) two values. Two constant identifiers are also defined: INTERRUPTABLE and UNINTERRUPTABLE. These constants should have appropriate values defined by the implementation. If the implementation augments this definition by adding extra priorities, extra constant identifiers should be placed in COROUTINES to denote each different priority.

A compilation unit may be given an explicit priority by a statement of the form

```
IMPLEMENTATION MODULE fred [COROUTINES.UNINTERRUPTABLE];
```

Note that the names of the constants denoting priority are defined by the standard but not the underlying type (which is defined by each implementation). The reason for this change to priority specifications is to allow implementations to provide whatever priority scheme is suitable - from PDP11-like levels (which can be implemented with numerical constants) to schemes where each device is individually specified (which cannot).

Two sample definitions of PRIORITY are provided in Appendix 1.

Any constant expression which defines a constant of an acceptable form is allowable in a priority specification, not only those constants defined by the system module COROUTINES. Local modules can also have an explicit priority specification.

Suitable operators on the type PRIORITY will be defined by the implementation. For example, if it is a numeric type, arithmetic operators will presumably be defined. It is open to the implementation to define no operators on the type: In that case, the constant expression denoting priority degenerates into one of the named constants provided in COROUTINES or another constant identifier which has been equated to one of them. Programmers may declare variables of type PRIORITY. Why they should want to is outside the scope of this document.

If the main program module specifies a priority, it will be run at that priority, otherwise it will inherit an implementation defined priority. The initialisation code of any module runs at the priority specified in the header or (if none is specified) at the priority applicable at the time of initialisation.

4.2. The initial priority of a Coroutine

If the module enclosing the body of a coroutine specifies a priority, the coroutine is to start at that priority. If no priority is specified, the coroutine will inherit the processor priority of the creating

[1] This rule also deals with the case where there are interjacent modules which do not specify the priority.

coroutine at the time of creation [2].

At first sight it seems reasonable to define a scheme without the notion of inheritance. The reason for including it comes from a desire to avoid the replication of software differing only in the priority specification: A high priority module is specified as such because the designer wishes to protect it from preemption whilst any invariant that the module maintains is temporarily invalid. It would be advantageous if it could use the services provided by other modules but, if a low priority coroutine is hidden inside such a service, an interrupt is possible. The provision of two modules differing only in Module name and priority (or N modules if there are N priorities) does circumvent this problem, albeit clumsily, without inheritance of priority. Another solution would be to code all service modules as Uninterruptable just in case they were called by an uninterruptable module. The subgroup preferred to extend the rules so that provision of several modules is not required, nor need they be defined at the highest priority.

A good example of the need for such a facility would be found in an attempt to provide the function of a CLU iterator [4] on a particular adt. A more specific example on the same lines is a routine which walks a tree, delivering one leaf node per call. If this were implemented by a low priority coroutine, any high priority module would have to ensure that invariants were satisfied before using it.

[2] Procedures defined within modules which specify a priority have to check that it will not be lowered (see discussion at the bottom of page 2). But there is a special case: A procedure which is the body of a coroutine must defeat the checking that would normally occur on the first TRANSFER to it. One possible way of achieving this is a one-time switch (on a per-coroutine basis) which is inspected by the priority checking code. Another, more radical, suggestion is to distinguish the bodies of coroutines from procedures by a new keyword.

5. Explicit Scheduling - Coroutines

The proposed standard for COROUTINES is modelled on the definition in PIM2. It has been given previously by Roger Henry in the paper N108 [2] presented to the open meeting in July 1986 in London. The VDM notation used here is defined by C.B. Jones [5].

5.1. Why change PIM?

The following is a summary of the discussion in N108. The initial desire of the group was to standardise the existing definition as it stood. We were dislodged from this position by a number of considerations:

Efficiency

A number of comments have been received on the potential inefficiency of linking the interrupt vector to the coroutine on each IOTRANSFER operation. We chose to separate the two for this reason and also because such a separation allows many different linking operations via different procedures without requiring many different versions of IOTRANSFER.

Clarity

It is argued that the original definition is hard to understand. Especially because a given Coroutine variable can refer to many different coroutines and because, although it can be assigned, it is by no means clear what such an assignment means (therefore don't do it?)

Machine specific

The definition is biased towards a PDP11.

The subgroup claim that the new definition is easier to understand, is closer to other definitions of coroutines and is not ambiguous.

5.2. How much code is affected?

Most people known to the author (except Randy Bush) do not "build a custom tasking model for each need" (N101) - they struggle to implement a module based on coroutines or use somebody else's, breathe a sigh of relief and use their chosen higher level procedures from then on. In many programs there will be no change (because they are not concurrent) or the change will be confined to one module. Even in programs which make a great use of explicit coroutine transfers, the conversion can be easily achieved by implementing the old version of TRANSFER etc. in terms of the new. Both this and a version of the new definition in terms of the old have been implemented by Don Ward and Roger Henry and are given in RH's paper N108.

5.3. The definition module of COROUTINES

Since COROUTINES is a system module and is not separately compiled, no definition module is needed. COROUTINES behaves as if the following were it's definition.

```
DEFINITION MODULE COROUTINES;

EXPORT QUALIFIED
       COROUTINE, NEWCOROUTINE,
       TRANSFER, IOTRANSFER,
       ATTACH, DETACH,
       SELF,
       PRIORITY,
       INTERRUPTABLE,
       UNINTERRUPTABLE;

TYPE
       COROUTINE,                  (* coroutine identity       *)
       PRIORITY = ... ;            (* Implementation defined *)

CONST
       INTERRUPTABLE = ...;        (* Implementation defined *)
       UNINTERRUPTABLE = ...;      (* Implementation defined *)

PROCEDURE NEWCOROUTINE
       ( Body: PROC;               (* code of coroutine     *)
         WorkSpace: ADDRESS;       (* base workspace address *)
         WsSize: CARDINAL;         (* workspace size (in storage units) *)
  VAR    cr: COROUTINE             (* identity of new coroutine *)
       );

PROCEDURE TRANSFER                 (* synchronous coroutine transfer *)
       ( To: COROUTINE             (* identity of destination coroutine *)
       );

PROCEDURE ATTACH                   (* Associate interrupt source with
                                      current coroutine *)
       ( Vector: CARDINAL          (* interrupt vector *)
       );

PROCEDURE DETACH ();               (* Disassociate current coroutine from
                                      sources of interrupts *)

PROCEDURE IOTRANSFER               (* synchronous out, asynchronous back *)
       ( To: COROUTINE;            (* identity of destination coroutine *)
  VAR    From: COROUTINE;          (* identity of interrupted coroutine *)
       );

PROCEDURE SELF                     (* deliver identity of current coroutine *)
       (): COROUTINE;

END COROUTINES.
```

5.4. The VDM definition of COROUTINES

The following is a sketch only. It falls to Derek Andrews to provide the full definition.

```
CID =                   (* Infinite set of Coroutine Id *)
CIDS =                  set of CID
Routines =              set of CID
DoingIO =               set of CID
Connected =             map CID to device
cc ∈ Routines           (* Current Coroutine *)
```

```
TRANSFER( d : CID )
ext  rd     Routines , DoingIO      wr  cc
pre         d ∈ Routines  ∧  d ∉ DoingIO
post        cc = swap(cc~,d)
```

```
IOTRANSFER( d : CID )
ext  rd     Connected , Routines    wr  cc , DoingIO
pre         cc ∈  dom Connected  ∧  d ∈ Routines
post        cc = swap(cc~,d)  ∧  DoingIO = DoingIO~ ∪ { cc~ }
```

```
ATTACH( d : device )
ext  rd     cc      wr  Connected
pre          true
post        Connected = Connected~ † { cc ↦ d }  ∧
            ¬( ∃c · ( Connected(c) = d  ∧  c ≠ cc ) )
```

```
DETACH
ext  rd     cc      wr  Connected
pre          true
post        Connected = { cc } ⩤ Connected~
```

```
INTERRUPT!(c : CID , d : device )
ext  rd     Connected       wr  cc , DoingIO
pre         c ∈  dom Connected  ∧  c ∈ DoingIO  ∧  Connected(c) = d
post        cc = swap(cc~,c)  ∧  DoingIO = DoingIO~ − { c } !!!!
```

NEWCOROUTINE( ) c : CID

ext	**rd**	CIDS	**wr** Routines
pre		**true**	
post		let $c \in CIDS - \overleftarrow{Routines}$	
		in $Routines = \overleftarrow{Routines} \cup \{c\}$	

swap(old , new : CID) : CID ≡ Space-for-Derek-to-draw-in

SELF( ) c : CID

ext	**rd**	cc
pre		**true**
post		c = cc

5.5. Returning from the body of a coroutine

The initialisation code of the main module is considered to have been called from the environment. Termination of the program occurs when an explicit or implicit return is made to this environment by the initialisation code of the main module.

On the other hand, a coroutine created within the program has no caller. An explicit or implicit return will cause an exception.

```
PROCEDURE Me():PROCESS;
PROCEDURE StopMe;
PROCEDURE NilProcess():PROCESS;

PROCEDURE StartProcess(
                    Body:       PROCESSBODY;
                    ExtraSpace: CARDINAL;
                    Param:      PARAMETER;
                    Urgency:    IMPORTANCE
                );
END PROCESSES.
```

6. Implicit Scheduling - Processes

The required separate module PROCESSES is specified by a definition module together with a description of each operation in VDM. When considering which functions to include in this module, the subgroup was motivated by economy: Our intent was to include only the minimum. Other more powerful, and perhaps more convenient, operations may be built on top of the ones we propose.

The essential difference between the facilities made available by this module and those provided by COROUTINES is the (assumed) indifference of the caller to which process is chosen next to run. Rather than explicitly choosing a successor by giving the destination as a parameter, the choice is made inside the module itself and may not be directly controlled by the user. It is of course possible to subvert the scheduling strategy and arrange that there is only one possible choice, but it is more honest and much easier to use COROUTINES directly if that is what is desired.

A further difference is that an implicit or explicit return is allowed from the body of a process - it has the same meaning as an explicit call to the termination routine StopMe.

The main process is composed of the initialisation code of all modules declared at level 0.

If the main process calls StopMe it will no longer be considered by the scheduler as a candidate to be the next to run. Under these circumstances there is no way for the program to terminate normally (see section 5.5): An exception may occur because of deadlock or other error conditions or the program will run continuously.

6.1. The definition module of PROCESSES

```
DEFINITION MODULE PROCESSES;

EXPORT QUALIFIED
        PROCESS, PARAMETER, PROCESSBODY, IMPORTANCE,
        SuspendMe, MakeReady, Me, StopMe,
        NilProcess, SuspendMeAndMakeReady, StartProcess,
        Associate, DisAssociate, SuspendUntilEvent;

TYPE
    PROCESS;       (* Is Opaque *)
    PARAMETER    = ADDRESS;
    PROCESSBODY  = PROCEDURE(PARAMETER);
    IMPORTANCE   = CARDINAL;


PROCEDURE SuspendMe;
PROCEDURE MakeReady(p: PROCESS);
PROCEDURE SuspendMeAndMakeReady(p: PROCESS);

PROCEDURE Associate(device: CARDINAL);
PROCEDURE DisAssociate;
PROCEDURE SuspendUntilEvent;
```

6.2. The VDM definition of PROCESSES

A process can be in one of three states: active, passive or waiting (for I/O completion). In addition there is usually one distinguished active process - the one that is currently running. All active processes are eligible to become the current process. Changing the current process is a scheduling operation which may be voluntary or involuntary. If there is no current process, either there is at least one waiting process or deadlock has occurred. Processes have an importance (or software priority) which influences the choice of process when scheduling takes place.

PID =	(* Infinite set of Process Id *)
PIDS =	**set of** PID
DEVICES =	**set of** device
Active $\subseteq$ PIDS	(* The set of processes which are eligible to be current i.e. the set of ready processes *)
Passive $\subseteq$ PIDS	(*The set of unready processes *)
Waiting $\subseteq$ PIDS	(*The set of processes waiting for I/O to complete *)
Attached =	**map** PID **to** device
Rank =	**map** PID **to** N
Current =	(* The current process *)

Invariant

$$\text{Is-Pairwise-disjoint}(\{\text{Active}, \text{Passive}, \text{Waiting}\}) \wedge$$
$$\text{Is-one-one}(\text{Attached}) \wedge$$
$$(\text{Current} = \textbf{nil} \vee \text{Current} \in \text{Active}) \wedge$$
$$\text{dom Rank} = \text{Active} \cup \text{Passive} \cup \text{Waiting} \wedge$$
$$\neg(\exists p \in \text{Active} \cdot \text{Rank}(p) > \text{Rank}(\text{Current}))$$

$$\text{deadlock} \triangleq \text{Active} \cup \text{Waiting} = \{\}$$

$$\text{Is-one-one}(m) \triangleq \textbf{card rng } m = \textbf{card dom } m$$

$$\text{Is-pairwise-disjoint}(S) \triangleq \forall x,y \in S \cdot x \neq y \Rightarrow x \cap y = \{\}$$

STARTPROCESS(Importance : N)
ext wr Active , Current , Rank
pre true
post let $p \in PIDS - (Active \cup Passive \cup Waiting)$
in $Active = \overleftarrow{Active} \cup \{p\} \wedge$
$(Rank(\overleftarrow{Current}) \geq Rank(p) \vee$
$(Rank(\overleftarrow{Current}) < Rank(p) \wedge Current = p))$
$Rank = \overleftarrow{Rank} \dagger \{p \mapsto Importance\}$

STARTPROCESS defines a new process and makes it active. It's process identity is not shared with any other process that is active, passive or waiting.

ME ( ) m : PID
ext rd Current
pre true
post m = Current

ME returns the process identity of the current process.

SUSPENDME
ext wr Active , Passive , Current
pre true
post $Active = \overleftarrow{Active} - \{\overleftarrow{Current}\} \wedge$
$Passive = \overleftarrow{Passive} \cup \{\overleftarrow{Current}\} \wedge$
Current = select(Active)

SUSPENDME makes the current process unready, ie. temporarily ineligible to be current. It is made eligible again (ready) by MAKEREADY

MAKEREADY (p : PID)
ext wr Active , Passive , Current
pre $p \in Passive$
post $Passive = \overleftarrow{Passive} - \{p\} \wedge$
$Active = \overleftarrow{Active} \cup \{p\} \wedge$
$(Rank(\overleftarrow{Current}) \geq Rank(p) \vee (Rank(\overleftarrow{Current}) < Rank(p) \wedge Current = p))$

It is an exception if the process to be made ready is not passive.

```
SUSPENDMEANDMAKEREADY ( p : PID )
ext   wr      Active , Passive , Current
pre           p ∈ Passive  ∨  p = Current
post          Active = ( ←Active − { ←Current } ) ∪ { p }  ∧
              Passive = ( ←Passive ∪ { ←Current } ) − { p }  ∧
              Current = select( Active )
```

SuspendMeAndMakeReady combines the two previous operations into a single indivisible operation. SuspendMeAndMakeReady(Me()) is a rescheduling operation which will cause a choice of which process to run from the most important active processes.

```
select( A : set of  PID ) cp : PID
ext   rd      Rank
pre           true
post          ( cp =  nil  ∧  A =  {} )  ∨  ¬( ∃p ∈ A · Rank( p ) > Rank( cp ) )

(* Possible implementation
        if  A =  {}
        then        cp =  nil
        else        let  p ∈ { pr ∈ A · ∀x ∈ A ( Rank( x ) ≤ Rank( pr ) ) }
                    in cp = p
*)
```

Select chooses a process from a given set. The selected process will be at least as important as any process in the set. Amongst sets of processes of equal rank the choice will be fair. The select operation is not directly callable.

```
ASSOCIATE( d : device )
ext   rd  Current       wr  Attached
pre           true
post          Attached = ←Attached † { Current ↦ d }  ∧
              ¬( ∃p · ( Attached( p ) = d  ∧  p ≠ Current ) )
```

ASSOCIATE attaches the current process to a device. Any other process that was attached to the device is no longer attached.

SUSPENDUNTILEVENT

ext wr	Active , Waiting , Current
pre	Current ∈ **dom** Attached
post	$Active = \overleftarrow{Active} - \{\overleftarrow{Current}\} \wedge$
	$Waiting = \overleftarrow{Waiting} \cup \{\overleftarrow{Current}\} \wedge$
	Current = select(Active)

SUSPENDUNTILEVENT suspends the process until the device, to which it is attached, signals completion of I/O by EVENT! .

EVENT! (p : PID , d : device)

ext wr	Active , Waiting , Current
pre	$p \in Waiting \wedge Attached(p) = d$
post	$Active = \overleftarrow{Active} \cup \{p\} \wedge$
	$Waiting = \overleftarrow{Waiting} - \{p\} \wedge$
	$(Rank(\overleftarrow{Current}) \geq Rank(p) \vee (Rank(\overleftarrow{Current}) < Rank(p) \wedge Current = p))$

Device d uses EVENT! to interrupt processing to signal that the I/O operation started by p requires attention. Note that only if Rank(p) > Rank(Current), will preemption occur. It is thus advisable for processes doing I/O to have a high importance if a quick response to the interrupt is wanted.

DISASSOCIATE

ext rd Current, **wr** Attached	
pre	Current ∈ **dom** Attached
post	$Attached = \{Current\} \lhd \overleftarrow{Attached}$

DISASSOCIATE detaches the current process from the device to which it was attached.

STOPME

ext wr	Active , Current
pre	**true**
post	$Active = Active - \{Current\} \wedge$
	$Current = select(Active) \wedge$
	$Rank = \{Current\} \lhd \overleftarrow{Rank}$
	$Attached = \{Current\} \lhd \overleftarrow{Attached}$

STOPME makes the process permanently ineligible for running. A new current process is chosen (if possible). The process will be detached from any device. If the body of a process returns, the effect will be as if a STOPME had been invoked.

7. Extra Facilities

It is worthwhile to outline the reasons why we have chosen to include semaphores and have chosen not to include monitors.

Why Semaphores?

Semaphores are well understood. They are an inherently procedural mechanism (cf. Monitors) and can be used to implement other synchronisation mechanisms. They are likely to be implemented by most people if not part of the standard environment.

Why not Monitors?

There are several subtly different definitions which deal differently with nested monitors. The mechanism is not inherently procedural; support is required from the compiler. Furthermore, the compiler also needs some knowledge of the facilities provided to arrange suspension of entrants whilst the monitor is occupied and reactivation when it is left. Thus PROCESSES, or a module like it, becomes almost part of the language. It is possible to implement a monitor procedurally, without compiler support [6] but it is then no more reliable than a semaphore - it is just as easy to miss out a call to ExitMonitor as it is to forget to release a semaphore.

7.1. Semaphores

7.1.1. The definition module of SEMAPHORES

```
DEFINITION MODULE Semaphores;
EXPORT QUALIFIED
                SEMAPHORE,
                Create, Destroy, Claim, Release;
TYPE
        SEMAPHORE;
PROCEDURE Create(
                VAR S: SEMAPHORE;
                InitialCount: CARDINAL
                );
PROCEDURE Destroy(
                 VAR S: SEMAPHORE
                );
PROCEDURE Claim(
                VAR S: SEMAPHORE
                );
PROCEDURE Release(
                VAR S: SEMAPHORE
                );
END Semaphores.
```

7.1.2. The VDM definition of SEMAPHORES

Each semaphore has a count associated with it and a set of processes waiting for it to become free. The semaphore is free if the count is non zero

SID = (* Infinite set of Semaphore Id *)
SIDS = set of SID
Count = map SID to N
Waiters = map SID to set of PID

Invariant

$$\forall s \in \text{dom Count} \cdot \text{Waiters}(s) \neq \{\} \Rightarrow \text{Count}(s) = 0 \ \wedge$$
$$\text{dom Count} = \text{dom Waiters}$$

CREATE (i : N) s : SID
ext rd SIDS wr Count , Waiters
pre true
post let $s \in \text{SIDS} - \text{dom Count}$
in $\text{Count} = \overleftarrow{\text{Count}} \dagger \{ s \mapsto i \} \ \wedge$
$\text{Waiters} = \overleftarrow{\text{Waiters}} \dagger \{ s \mapsto \{\} \}$

CREATE defines a new semaphore. The count is initialised to the parameter given. No process is waiting for it to be free.

DESTROY (s : SID)
ext wr Count , Waiters
pre $s \in \text{dom Count} \ \wedge \ \text{Waiters}(s) = \{\}$
post $\text{Count} = \{ s \} \lhd \overleftarrow{\text{Count}} \ \wedge$
$\text{Waiters} = \{ s \} \lhd \overleftarrow{\text{Waiters}}$

An attempt to destroy a semaphore on which there are processes waiting will cause an exception.

CLAIM (s : SID)

ext wr Count , Waiters

pre $s \in$ dom Count

post $(\text{Count}(s) = \overleftarrow{\text{Count}(s)} - 1 \wedge \overleftarrow{\text{Count}(s)} > 0) \vee$
$(\text{Waiters}(s) = \overleftarrow{\text{Waiters}(s)} \cup \overleftarrow{\text{ME}} \wedge \text{post-SUSPENDME})$

If the count associated with the semaphore is non zero it is decremented, otherwise the current process is suspended and added to the set waiting for it to become free.

RELEASE (s : SID)

ext wr Count , Waiters

pre $s \in$ dom Count

post $(\text{Count}(s) = \overleftarrow{\text{Count}(s)} + 1 \wedge \overleftarrow{\text{Waiters}(s)} = \{\}) \vee$
$(\overleftarrow{\text{Waiters}(s)} \neq \{\} \wedge$
let $p = \text{select}(\overleftarrow{\text{Waiters}(s)})$
in post-MAKEREADY(p) $\wedge$ $\text{Waiters}(s) = \overleftarrow{\text{Waiters}(s)} - \{p\}$
)

The Notation "$\text{Count}(s) = \overleftarrow{\text{Count}(s)} + y$" is an abbreviation for "$\text{Count} = \overleftarrow{\text{Count}} \dagger \{ s \mapsto \overleftarrow{\text{Count}(s)} \}$"

If no process is waiting for the semaphore, the count associated with it is incremented, otherwise one process is selected from those waiting for it, removed from the waiting set and made eligible to run. Preemption will occur if the newly eligible process is more important than the current process.

References

1 N. Wirth "Programming in Modula-2"
Springer-Verlag 1982, 1983, 1985

2 R. Henry "Coroutines and Processes" (N108)

3 J. Welsh, P. Bailes "The Go-betweens' tale" (N105)

4 B Liskov, J Guttag "Abstraction and specification in program development"
MIT Press 1986

5 C.B. Jones, "Systematic software development using VDM".
Prentice-Hall 1986

6 P.D. Terry, "A Modula-2 Kernel for Supporting Monitors"
Software: Practice and Experience 16(5) pp 457-472 (May 1986)

8. Appendix 1 - Sample specifications of PRIORITY

```
DEFINITION MODULE COROUTINES;

EXPORT QUALIFIED
        ....... ,  (*       NEWCOROUTINE etc.    *)
        PRIORITY,
        UNINTERRUPTABLE,
        HIGH_PRIORITY,
        MED_PRIORITY,
        LOW_PRIORITY,
        INTERRUPTABLE;
TYPE
        PRIORITY                = [0 .. 7];
CONST
        UNINTERRUPTABLE     = 7;
        HIGH_PRIORITY       = 5;
        MED_PRIORITY        = 3;
        LOW_PRIORITY        = 1;
        INTERRUPTABLE       = 0;
        .
        .
END COROUTINES.

DEFINITION MODULE COROUTINES;

EXPORT QUALIFIED
        ....... ,  (*       NEWCOROUTINE etc.    *)
        UNINTERRUPTABLE,
        F, P, K, S, FP, FK, FS, PK, PS, KS, FPK, FPS, FKS, PKS,
        INTERRUPTABLE;
TYPE
        Device = (Floppy,Printer,Keyboard,Screen);
CONST
        INTERRUPTABLE       = Device {};
        F                   = Device { Floppy };
        P                   = Device { Printer };
        K                   = Device { Keyboard };
        S                   = Device { Screen };
        FP                  = F + P;
        FK                  = F + K;
        FS                  = F + S;
        PK                  = P + K;
        PS                  = P + S;
        KS                  = K + S;
        FPK                 = F + P + K;
        FPS                 = F + P + S;
        FKS                 = F + K + S;
        PKS                 = P + K + S;
        UNINTERRUPTABLE     = F + P + K + S;
        .
        .
END COROUTINES.
```

9. Appendix 2 - An example use of PROCESSES and SEMAPHORES

The following module is a simple example which demonstrates the use of semaphores and processes. The module provides a facility similar to the COBEGIN...COEND construct of concurrent Pascal.

9.1. CoBEGIN .. CoEND

After a CoBEGIN call, the parent process may create children by calling the CoStart procedure. All the children wait until the parent executes the CoEND procedure. The parent then waits until all the children are finished: They finish by returning from the procedure given as a parameter to CoStart.

Inside the module, two semaphores are used: One to provide mutual exclusion between parents whilst procreating and one between children when terminating. The use of these two semaphores avoids interesting program behaviour caused by race conditions. All the starting and finishing protocol is in a hidden procedure which provides a pre and postlude to the user's code.

A further pair of semaphores (one pair per CoOperating family) is used to exclude children whilst the parent is creating the rest of the family and to exclude the parent whilst the children are running. The module could be written with only one semaphore for both purposes, since the critical sections follow one another sequentially, but using two leads to clearer code.

Having one or more children call Processes.StopMe (rather than returning from the procedure which is specified as the body of the child) is an excellent way to deadlock the program. The parent will remain suspended for ever because the exit protocol, which activates the parent when all the children have finished, will not be obeyed. Modifying the module to remove this (mis)feature is left as an exercise for the reader.

9.1.1. The definition module of CoBEGIN .. CoEND

```
DEFINITION MODULE CoOp;

FROM Processes      IMPORT      PROCESS, PROCESSBODY, PARAMETER, IMPORTANCE;
EXPORT QUALIFIED
            CB,
            CoBEGIN, CoStart, CoEND;
TYPE
      CB;

PROCEDURE CoBEGIN(
            VAR Block: CB
            );

PROCEDURE CoStart(
            c:          CB;
            UserProc:   PROCESSBODY;
            Space:      CARDINAL;
            UserParam:  PARAMETER;
            Urgency:    IMPORTANCE
            );
```

PROCEDURE CoEND(
VAR Block: CB
);
END CoOp.

9.1.2. The VDM definition of CoBEGIN .. CoEND

CoBEGIN .. CoEND is modelled by two maps. Children(p) are the processes created by process p using the COSTART operation. Finished(p) are the children of p who have finished execution.

No process is its own child. No process is the child of more than one parent. No more children can finish than have been created.

Children = **map** PID **to** **set of** PID
Finished = **map** PID **to** **set of** PID

Invariant
$\forall p \in$ **dom** Children ($p \notin$ Children(p) $\wedge$ Finished(p) $\subseteq$ Children(p) $\wedge$
$\forall y \in$ **dom** Children ($p = y \vee$ Children(p) $\cap$ Children(y) $= \{\}$)
)

COBEGIN
ext **wr** Children , Finished
pre ME $\notin$ **dom** Children
post Children = $\overleftarrow{\text{Children}}$ † { ME $\mapsto$ {} } $\wedge$
Finished = $\overleftarrow{\text{Finished}}$ † { ME $\mapsto$ {} }

COBEGIN indicates that the current process is about to create children. None have yet been created and none have finished execution.

COSTART (Importance : N)
ext **rd** Active , Waiting , PIDS **wr** Passive , Rank , Children
pre ME $\in$ **dom** Children
post **let** p $\in$ PIDS $-$ (Active $\cup$ Passive $\cup$ Waiting)
in Passive = $\overleftarrow{\text{Passive}} \cup$ { p } $\wedge$
Rank = $\overleftarrow{\text{Rank}}$ † { p $\mapsto$ Importance } $\wedge$
Children (ME) = $\overleftarrow{\text{Children(ME)}} \cup$ { p }

COSTART defines a new process which is not eligible for running. The new process is a child of the current process

COEND
ext **rd** Waiting **wr** Active , Passive , Children , Finished
pre ME ∈ **dom** Children
post $\neg$ ($\exists$p ∈ Children($\overleftarrow{ME}$) · p ∈ Active ∪ Passive ∪ Waiting) ∧
Children = $\overleftarrow{ME}$ ⩤ $\overleftarrow{Children}$ ∧
Finished = $\overleftarrow{ME}$ ⩤ $\overleftarrow{Finished}$
(* Possible implementation
if Children (ME) = {}
then I
else Active = $\overleftarrow{Active}$ ∪ Children ($\overleftarrow{ME}$) ∧
Passive = $\overleftarrow{Passive}$ − Children ($\overleftarrow{ME}$) ∧
post-SUSPENDME
*)

COEND makes the children of the current process active and suspends the current process until all of the children have finished execution. On completion, all children will not exist. Each child will, at the end of it's execution obey a wrap-up operation. No more children may be created until another COBEGIN operation.

wrap-up
ext **rd** Children **wr** Finished
pre ME ∈ **rng** Children
post **let** Children (parent) = $\overleftarrow{ME}$
in Finished (parent) = $\overleftarrow{Finished(parent)}$ ∪ { $\overleftarrow{ME}$ } ∧ post-STOPME ∧
(Finished (parent) = Children (parent) ∧ post-MAKEREADY(parent)
(Finished (parent) ⊂ Children (parent) ∧ parent ∈ Passive)

If all the children have finished (ie. this is the last child to wrap-up) the parent process will become active and will complete the COEND operation. In any case the child is permanently ineligible for running. The wrap-up operation is invoked implicitly when the child comes to the end of or returns from it's procedure body.

9.1.3. The implementation of CoBEGIN .. CoEND

```
IMPLEMENTATION MODULE CoOp;

FROM  Semaphores    IMPORT    SEMAPHORE,
                              Create, Destroy, Claim, Release;
FROM  Processes     IMPORT    PROCESS, PROCESSBODY, PARAMETER, IMPORTANCE,
                              StartProcess;
FROM  Storage       IMPORT    ALLOCATE, DEALLOCATE;

TYPE  CoBlock =     RECORD
                        ChildPen:     SEMAPHORE;
                        ChildMinder:  SEMAPHORE;
                        Children:     CARDINAL;
                    END;
      CB      =     POINTER TO CoBlock;

VAR
      Parenthood:   SEMAPHORE;
      Childhood:    SEMAPHORE;

      Proc:         PROCESSBODY;(* Used to smuggle extra params  *)
      CurrentCb:    CB;         (* between parent and child     *)

CONST
      CoOpTariff =  200;    (* Extra space for CoOperating processes *)
                            (* IMPLEMENTATION DEPENDENT              *)

PROCEDURE CoBEGIN(VAR c: CB);
BEGIN
      ALLOCATE(c, SIZE(CoBlock));   (* come back NEW, all is forgiven *)

      Create(c^.ChildPen, 0);       (* both semaphores are created    *)
      Create(c^.ChildMinder, 0);    (* initially claimed              *)
      c^.Children := 0;
END CoBEGIN;
```

```
PROCEDURE CoEND(VAR c: CB);
VAR i:  CARDINAL;
BEGIN
        IF c^.Children > 0
        THEN
                FOR i := 1 TO c^.Children
                DO
                        Release(c^.ChildPen);
                END;
                (*      Wait for children to finish*)
                Claim(c^.ChildMinder);
        END;
        (*      All children now finished *)
        Destroy(c^.ChildPen);
        Destroy(c^.ChildMinder);
        DEALLOCATE(c,SIZE(CoBlock));
END CoEND;


PROCEDURE Wrapping(p: PARAMETER);
VAR     UserProc:       PROCESSBODY;
        c:              CB;
BEGIN
        (*      Copy params to local space while there is no            *)
        (*      interference from other parents ( because of claimed    *)
        (*      Parenthood semaphore)                                   *)

        UserProc := Proc;
        c := CurrentCb;
        Release(Parenthood);    (*      Allow other progeny             *)
        Claim(c^.ChildPen);     (*      Wait until parent CoENDs        *)
                                (*      and then ....                   *)
        UserProc(p);            (*      Call the punter's code          *)

        Claim(Childhood);
        DEC(c^.Children);       (*      These two operations must be    *)
        IF c^.Children = 0      (*      indivisible                     *)
        THEN
                (*      Last Child ... wake parent*)
                Release(c^.ChildMinder);
        END;
        Release(Childhood);
END Wrapping;
```

```
PROCEDURE CoStart(
                c:          CB;
                UserProc:   PROCESSBODY;(* Extra params are     *)
                Space:      CARDINAL;   (* as for Startprocess  *)
                UserParam:  PARAMETER;  (* and have the same    *)
                Urgency:    IMPORTANCE  (* meaning              *)
                );
BEGIN
        Claim(Parenthood);
        Proc := UserProc;
        CurrentCb := c;
        INC(c^.Children);
        StartProcess(Wrapping, Space + CoOpTariff, UserParam, Urgency);
END CoStart;

BEGIN
        Create(Parenthood, 1);  (* Allow one parent          *)
        Create(Childhood, 1);   (* and one child             *)
                                (* through critical sections *)
END CoOp.
```

Modula-2 News Issue # 0 October 1984

Modula-2 News Issue # 1 January 1985

Review of Gleaves' Modula-2 text by Tom DeMarco
MODUS Paris meeting 20/21 Sep 84, C.A. Blunsdon
Report of M2 Working Group, 8 Nov 84, John Souter
Modula-2 Standard Library Rationale, Randy Bush
Modula-2 Standard Library Definition Modules
Modula-2 Standard Library Documentation, Jon Bondy
Validation of M2 Language Implementations, J. Siegel

MODUS Quarterly # 2 April 1985

Letters, Anderson & Emerson
Opaque Types in Modula-2, C. French & R. Mitchell
Dynamic Module Instantiation, Roger Sumner
The Linking Process in Modula-2, Jeanette Symons
Modula-2 Library Comments, Bob Peterson
Modula Compilers - Where to Get 'em, Larry Smith
Coding War Games Prospectus, Tom DeMarco
M2, An Alternative to C, M. Djavaheri, S. Osborne

MODUS Quarterly # 3 July 1985

Letters, Endicott & Hoffman
Some Thoughts on Modula-2 in "Real Time", Paul Barrow
RajaInOut: simple, safer, I/O for
Logitech/MS-DOS, R. Thiagarajan
Selection of Contentious Problems, Barry Cornelius
Expressions in Modula-2, Brian Wichmann
The Scope Problems Caused by Modules, Barry Cornelius

MODUS Quarterly # 4 November 1985

State of MODUS, George Symons
MODUS Meeting Report, Bob Peterson
A Writer's View of a Programmer's Conference, Sam'l Bassett
Concerns of A programmer, Dennis Cohen
Modifications to the Standard Library
Proposal, R. Nagler & J. Siegel
Proposal, standard library and M2 extension,
Odersky, Sollich, & Weisert
Standard Library of the Unix OS, Morris Djavaheri
The Standard Library for PC's, E. Verhulst
Editorial, Richard Karpinski
Modula-2 Compilation and Beyond, D.G. Foster
Modula-2 Processes - Problems and Suggestions, Roger Henery

MODUS Quarterly # 5 February 1986

Editorial, Richard Karpinski
Exporting a Module Identifier, Barry Cornelius
Letter on multi dimensional open arrays, Niklaus Wirth
Letter on DIV, MOD, /, and REM, Niklaus Wirth
BSI Accepted Change: Multi-dim. open arrays, Willy Steiger
N73: NULL-terminated strings in Modula-2, Ole Poulsen
ISO Ballot Results re BSI Specifying Modula-2
Draft BSI Standard I/O Library for Modula-2, Susan Eisenbach
Portable Language Implementation Project: Design and
Development Rationale, K. Hopper and W.J. Rogers
The ETH-Zuerich Modula-2 for the Macintosh, Chris Jewell
NewStudio: Engineering a Modula-2 Application for the Mac,
A. Davidson, H.B. Herrmann, E.R. Hoffer

MODUS Quarterly # 6 November 1986

Editorial, Richard Karpinski
Letter on opaque types, File type, and SET OF CHAR, P. Williams
Letter on exported identifiers, E. Videki
Why the Plain Vanilla Linkers, J. Gough
Letter re best article & MacModula-2, M. Coren
Significant Changes to the Language Modula-2, Barry Cornelius
All About Strings, Barry Cornelius
Type Conversions in Modula-2, B. Wichmann
Improving the quality of Definition Modules, A. Sale
A Programming Environment for Modula-2, F. Odegard
Academic Modula-2 Survey, L. Mazlack
Compilers for Modula-2 (Zuerich list)
Membership List

MODUS Quarterly # 7 February 1987

Editorial, Richard Karpinski
New Products
Modula-2 Standardisation: A go betweens tale, Welsh & Bailes
Modula-2 VM/CMS, Thomas Habernoll
TCP Implementation in Modula-2, F. Ma & L. D. Wittie
Building an Operating System with Modula-2,
B. Justice, S. Osborne, & V. Wills
Note on Implementing SET OF CHAR, Source Code
for a SetOfChar MODULE, A. Brunnschweiler

MODUS Administrators supply single copies at $5 US of 12 Swiss Francs.

Hints for contributors:

Send CAMERA READY copy to the editor (dot matrix copy is usually unacceptable). Machine readable copy is preferred. Present facilities permit printing from electronic mail and floppy disks (Sage, IBM PC, Macintosh) using Postscript, Script, TeX, and troff formatting systems. Working papers and notes about work in progress are encouraged. MODUS Quarterly is not perfect, though it tries to be current.

Please indicate that publication of submission is permitted.
Correspondence not for publication should be PROMINENTLY so marked.

Send your submissions to:

Richard Karpinski, Editor	TeleMail	M2News or RKarpinski
6521 Raymond Street	BITNET	dick@ucsfcca
Oakland, CA 94609	Compuserve	70215,1277
(415) 476-4529 (12-7 pm)	InterNet	dick@cca.ucsf.edu
(415) 658-3797 (ans. mach.)	UUCP	...ucbvax!ucsfcg!cca.ucsf!dick

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : _____

Affiliation : _____

Address : _____

Address : _____

City : _____

State : _____ **Postal Code:** _____ **Country:** _____

Phone : (___)___-_____ **Electronic Addr :** _____

Application as: **New Member** _____ **or Renewal** _____

Implementation(s) used : _____

Option: ___ Do NOT print my phone number in any rosters
or: ___ Print ONLY my name and country in any rosters
or: ___ Do NOT release my name on mailing lists

** Membership fee per year (20 USD or 45 SFr) **

Members of the US group who are outside of North America, add $10.00.

In North and South America, please send check or money order (drawn in US dollars) payable to Modula-2 Users' Association at:

Modula-2 Users' Association
P.O. Box 51778
Palo Alto, California 94303
United States of America

Otherwise, please send check or bank transfer (in Swiss Francs) payable to Modula-2 Users' Association at:

Aline Sigrist
MODUS Secretary
ERDIS SA
Postfach 35, CH-1800 Vevey 2
Switzerland

The Modula-2 Users' Association is a forum for all parties interested in the Modula-2 Language to meet each other and exchange ideas. The primary means of communication is through the Newsletter which is published four times a year. Membership is for an academic year, and you will receive all newsletters for the full year in which you join. Mid-year applications receive that year's back issues. Modula-2 is a new and developing language; this organization provides implementors and serious users a means to discuss and keep informed about the standardization effort, while discussing implementation ideas and peculiarities. For the recreational user, there is information on the status of the language, along with examples and ideas for programming in Modula-2. For everyone, there is information on current implementations and the other resources available for obtaining information on the language.

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : _____

Affiliation : _____

Address : _____

Address : _____

City : _____

State : _____ **Postal Code:** _____ **Country:** _____

Phone : (___)____-_____ **Electronic Addr :** _____

Application as: **New Member** _____ **or Renewal** _____

Implementation(s) used : _____

Option: ___ Do NOT print my phone number in any rosters
or: ___ Print ONLY my name and country in any rosters
or: ___ Do NOT release my name on mailing lists

** Membership fee per year (20 USD or 45 SFr) **
Members of the US group who are outside of North America, add $10.00.

In North and South America, please send check or money order (drawn in US dollars) payable to Modula-2 Users' Association at:

Modula-2 Users' Association
P.O. Box 51778
Palo Alto, California 94303
United States of America

Otherwise, please send check or bank transfer (in Swiss Francs) payable to Modula-2 Users' Association at:

Aline Sigrist
MODUS Secretary
ERDIS SA
Postfach 35, CH-1800 Vevey 2
Switzerland

The Modula-2 Users' Association is a forum for all parties interested in the Modula-2 Language to meet each other and exchange ideas. The primary means of communication is through the Newsletter which is published four times a year. Membership is for an academic year, and you will receive all newsletters for the full year in which you join. Mid-year applications receive that year's back issues. Modula-2 is a new and developing language; this organization provides implementors and serious users a means to discuss and keep informed about the standardization effort, while discussing implementation ideas and peculiarities. For the recreational user, there is information on the status of the language, along with examples and ideas for programming in Modula-2. For everyone, there is information on current implementations and the other resources available for obtaining information on the language.

MODUS
c/o Pacific Systems
P.O.Box 51778
US-Palo Alto, CA 94303

MODUS
c/o ERDI S.A.
Case Postale 35
CH-1800 Vevey 2