MODUS QUARTERLY ISSUE NO. 4 NOVEMBER 1985

The MODUS Quarterly

Issue # 4

November 1985

>> YOUR LAST ISSUE <<

until you renew.

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

CONTENT

Directors of MODUS, the Modula-2 Users Association:

Randy Bush
Pacific Systems Group
601 South 12th Court
Coos Bay, OR 97420
(503) 267-6970

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

Jean-Louis Dewez
Laboratoire de Micro Informatique
Conserveratoire ANM
2, Rue Conte
F-75003 Paris
(01) 271-2414

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

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

Administration and membership:

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

Europe: Mrs. Aline Sigrist
c/o ERDIS
Case Postale 35 (021) 52 61 71
1800 Vevey 2 Switzerland

Editor, Modula-2 News:

Richard Karpinski
6521 Raymond Street
Oakland, CA 94609

Weekdays	(415) 666-4529 (12-7 pm)
Anytime	(415) 658-3797 (ans. mach.)
TeleMail	M2News or RKarpinski
BITNET	dick@ucsfcca
Compuserve	70215,1277
USENET	...!ucbvax!ucsfcgl!cca.ucsf!dick

>> Problems? Missing an issue? <<

Contact your membership coordinator (see above).

Publisher:

George Symons (see above)

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Correspondence not for publication should be PROMINENTLY so marked.

Publication schedule:

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

RENEWAL NOTICE

We have now reached Issue Number 4 of the Newsletter. This means that over one year has transpired since MODUS began accepting memberships and printing Newsletters. This also means that it is time for everyone to renew their membership for another year so that they may continue receiving the Newsletters. This is done by filling out a new Membership Application (which can be found in this Newsletter), and sending it in with a check to the appropriate address.

For those who are confused by this renewal policy, it is because Modus membership is on an academic year basis. When you send in your money and membership form, you are a member for that academic year (including the following summer), and therefore receive all of the Newsletters for that year (including those missed if you join mid-year) This both makes it easier on the organization since renewals happen only once a year, and provides the members with all of the Newsletters for that year so that it is harder to miss information.

During this year, membership in MODUS has grown to over 300 members through the United States office, and over 200 members through the Switzerland office. We are looking forward to continued growth during this next year, with the following goals in mind:

- Publish more papers in the Newsletter.
- Publish more language and library discussions
- Hold the September and possibly future meetings
- Cooperate further with the British Standards Institution in open meetings

We look forward to your continued membership.

Sincerely,

George J. Symons
Secretary

LAST CHANCE

PLEASE RENEW NOW

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : ____

Affiliation : ____

Address : ____

Address : ____

State : ______________ **Postal Code** : ________ **Country** : ______________

Phone : (____) ____ - __________ **Electronic Addr** : ______________

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

Application as **New Member** ___ **or** **Renewal** ___

Implementation(s) Used ______________________________________

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

P.O Box 51778
Palo Alto, California 94303
United States

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

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

The Modula-2 Users' Association is a forum for all parties interested in the Modula-2 Language to meet 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 will be information on the status of the language, along with examples and ideas for programming in Modula-2. For everyone, there is information on current known implementations and other resources available for information on the language.

The State of Modus

Welcome to a new year! Modus has successfully completed its first year of publishing newsletters, and except for a slight delay with the final issue, I think we did an excellent job. I would like to thank all of you who contributed to the newsletter, and to encourage those of you who have not to please submit an article or a letter if you feel the subject is important to you.

Modus has also managed to sponsor a number of meetings during the year. Two of these were in the UK and Europe, with the final one being held in the USA. I only attended the meeting in the US, but from the reports I received, all of these meetings were quite successful and deserve a repeat performance sometime during this coming year.

During our initial year, the organization evolved in many ways, and grew to a size larger than our original projections. For those of you who are not familiar, Modus is a diverse international organization. We have headquarters in Palo Alto, California and Vevey, Switzerland. The organization shares the newsletter between the two headquarters as well as many mutual directors. But legally, there are two separate organizations, with the US portion being a California Corporation.

The membership of Modus has grown from just a handful at the beginning of last year to over 350 in the US and over 250 in Europe with the number growing every day. We have received a lot of cooperation from the companies producing and selling Modula-2 compilers which has helped spread information on the organization without requiring advertising by Modus. Modus does not plan to advertise in order to increase membership. We are interested in providing a means of communication for serious users and implementors of Modula-2, and information for those interested in the language. To meet this goal, compiler vendors notices, and word of mouth from you gets Modus information to potential members.

As far as the financial situation, I only have figures for the California Corporation. The Corporation is solvent and even slightly profitable. With income from Memberships at about $7,000 and payment from the European group for Newsletters that were sent over in bulk, there was a $950 profit at the end of our fiscal year (Sept. 30). There are also about 400 copies of each issue of the Newsletter left over for new members who want them.

I would like to take this opportunity to thank all of the people who have given their time to help Modus. Modus has been run completely by volunteers. Up to this time the only services we have paid for are our legal, accounting and printing services. I would like to give special thanks to Randy Bush and Tom DeMarco who spearheaded the effort on the US side to make the organization a reality. I would also like to thank all of the people at Pacific Systems Group who gave their time to make the make sure the membership database was in order, and the first three issues of the Newsletters were mailed out. I would also like to thank Logitech and Charmaine Bennett for taking over the job of mailing out the Newsletter. And most of all, I would like to thank Dick Karpinski for being the editor of this Newsletter. [Being is easy, doing is hard. rhk

I hope all of last years members enjoyed the Newsletters and received information that was useful to you. I also hope that the information that will be upcomming in this and future issues will be at least valuable.
But remember, that depends on you.

George J. Symons, Secretary

MODUS Meeting Report
by Bob Peterson

This article is about the recent MODUS meeting held in Menlo Park, CA, Thursday and Friday, September 5 & 6. I don't intend to cover much in way of content, since other articles will contain detailed information. Rather, I hope to provide something of the flavor or atmosphere of the meeting. To do so, I will adopt something of an editorial point of view, ie, writing about my impressions more than the "facts".

The Alhambra Conference Center, site of the meeting, is like an oasis in the middle of the desert. The Center grounds featured large, old evergreen trees, lots of lawns, vines on trellises, well-kept buildings, and an old mansion with an iron front door. Sleeping rooms were not luxurious, but quite adequate. The meals, served in the cafeteria to all attendees, were quite good. And since everyone ate together there was lots of good conversation during and after each meal.

Attendees began arriving Wednesday evening in time for dinner. Afterward almost everyone participated in an informal social gathering in the mansion.

Thursday morning the formal program began with introductions, schedule corrections, and a bit of Modus' history, current status, and a minimum of formal business.

Much of the morning was taken up by Susan Eisenbach's discussion of the current British Standards Institute's efforts to standardize Modula 2. Ms. Eisenbach covered the Committee's organization, introduced the members present, and summarized the current state of the effort and the schedule leading to adoption. A number of questions were asked by attendees, leading some some discussion of how the Committee was proceeding.

Immediately after lunch, Ms. Eisenbach continued to be the center of attention with a presentation titled, "Contentious Language Issues". While summarizing issues on which the Committee has taken a position, the discussion focused on language issues which are not clear and which remain unresolved. As might well be expected, many questions and comments came from the audience, and Ms. Eisenbach, as the Committee's representative, asked for a sense of where the audience members stood on some of the issue

After the "Contentious Issues" discussion died down, Mr. Ed Videki explained "An Often Mis-implemented Aspect of the Language". The issue Mr. Videki highlighted relates to scoping rules and, in detail, how various declarations are or are not visible in various scopes.

The scheduled "Compiler Implementors' Panel" session, scheduled to follow Mr. Videki, wasn't really a panel, but presentations by the various compiler implementors present at the meeting. Due to time constraints, some of these presentations were slipped into Friday's schedule.

Presenters included Ken Butler of Tartan Labs on their VAX/Unix compiler, Morris Djavaheri of Djavaheri on their 68000/Unix compiler for various Unix flavors, Eberhart Enger on his CP/M-68K adaptation of the Zurich 68000 sources, David Foster of CERN discussing the adaptation done by and for the high energy physics researchers, Peter Sollich and Mike Weisert discussed Borland's Z80/CP/M-80 compiler and library, Giacomo Marini of Logitech talking about current and future Modula 2 products, and Jeff Savit of Savvy Computing talking about a compiler for the 370/CMS environment.

The CERN Modula 2 compiler system includes at least one tool not generally provided: utilities to create and maintain a database of module versions, to assist the development team. The utilities track which version of a module is current, point out which object modules are invalid, and help create compiler scripts. Unfortunately, the compiler and utilities are not products and are not generally available.

The Borland presentation discussed the Z80/CP/M compiler, but did mention that a PC-DOS/MS-DOS/8086 version is also in the works. While not yet a product, an early copy of the functioning system (compiler, editor, and execution control) was available for demonstrations. Also made available were copies of Borland's library implementation, including some documentation on language extensions. See the article for details.

Jeff Savit offered some very interesting comments about moving Modula 2 into a mainframe environment using a different character set (EBCDIC) and batch processing. Savvy Computing is just getting started on this effort.

Most of the compilers process the language described in _Programming in Modula 2 Second (corrected) Edition_. A Third Edition is now available, as well as letters documenting other language changes Dr. Wirth has made. The BSI draft standard will also be available in the near future. Several of the compiler writers are eager to have a stable language to process as well as a "good" standard libary.

Library issues began the Friday discussions. Randy Bush, a BSI Committee member, introduced the topic and discussed the Ad Hoc Library Committee's efforts from a philosophical and stylistic point of view rather than describing the technical aspects of the proposal. Most of the attendees had at least read the proposal in Modus News Issue #1, January 1985, and the comments in the following issue.

Like the Compiler Implementor's Panel, the Library Implmenetor's Panel wasn't really a panel, but more of a series of interactive presentations. Roger Henry of the University of Nottingham and a BSI Committee member discussed processes and where in the language they should be. Comparing Modula-1 processes with Modula-2 processes, a number of problems with Modula-2 processes can be seen. Providing a portable implementation of coroutines (the language construct underlying processes) seems to be best done by standardizing the SYSTEM module or by moving process-related facilities into the language as standard identifiers. See his paper for details.

Rob Nagler, implementor of several versions of the Ad Hoc Committee's proposed library, made a number of observations. Other than style comments, he felt some of the modules were improperly structured and, his major comment, that too many decisions were left to the implementor. He also offered his own Standard Library Proposal and asked for comments on it. [The full paper is available from Dick Karpinski at cost. rhk]

The final session allowed users of Modula-2, as opposed to implementors, to discuss how they use the language, problems they've had, and so forth.

Terry Anderson, from Walla Walla College, has moved a simulation package from mainframes to micros, using UCSD Pascal, and has recently moved much of the package into Modula 2.

Bill Bonham from Stride Micro discussed an operating system he is writing in Modula 2. So far there is a very small part that had to be written in assembler. He's using a compiler supplied by Scenic Computer, which is based on work done by Volition Systems. The OS assumes no management hardware, multitasking, multiple terminals of varying brand multiple jobs running from a single terminal.

Stan Osborne, San Francisco State University, and two of his students described how the kernel of an operating system was developed in Modula The interesting wrinkle was the design approach they used. It avoids contention and unnecessary waiting on resource locks better than other design approaches.

Richard Pattis discussed teaching introductory computer science classes using Modula 2. His major problem is lack of textbooks. The major issue he faces is what sequence to use to introduce the various programming concepts.

The meeting wound up with closing remarks by Randy Bush.

[We save a whole page by starting the Sam'l Basset article here. rhk]

A Writer's View of a Programmer's Conference
by Samuel Basset

Being a computer-literate liberal arts graduate, rather than an engineer or programmer, I found the attitudes and outlooks of the people at the MODUS meeting on September 5 and 6 as interesting as the contents of the presentations.

The most striking dichotomy was between theoreticians and practitioners -- most of the former were European, and seemed to be most interested in the purity, clarity, and unambiguousneylss of the definition of the language. Most of the latter were denizens of the U.S.A., and were most concerned with getting something "out there" that _worked_, and cleaning up the details later, when the finicky details of language definition had been worked out (by someone else).

In response to practitioners' growls from the back of the room: "How are you going to get _that_ to work on a _real_ computer?", the standard theoretical response seemed to be: "Well, we will face that when we come to it -- it's merely a matter of implementation, after all -- but we _must_ have a clear definition of the language, to prevent its fragmentation into a host of dialects, like Pascal."

Both sides were reasonably civil to one another -- most of the theoreticians, being British, were very good at that -- and I do not remember any ad hominem attacks or real acrimony. The practitioners grudgingly admitted that theoretical purity would be very nice to have, but pointedly drew the group's attention to the need for speed -- getting working compilers into the hands of ordinary programmers as soon as possible. They also expressed some doubt as to whether there would _ever_ be a complete and thoroughly accepted standard.

The other dichotomy was between what I would call the organizationals and the entrepreneurs. The dividing line between these two groups was clearly _not_ the Atlantic Ocean, however.

The organizational interest in a Modula Language Standard seemed to be threefold: consensus, marketing and support.

The consensus point of view, which is explicitly the position of the British Standards Institute team, was that as wide agreement as possible to the Standard before promulgating even a proposed draft is vitally necessary -- evidently to avoid later backbiting of the "But you didn't address the issue of . . .") sort.

[For copies of the BSI Working Documents, contact Karpinski.]

The marketing people want a Standard as quickly as possible, so that they can point to it, and say: "See -- we have a Standard Modula, which, if you buy it, will do all of these wonderful things for your programming effort."

The support people are pretty blunt about saying that a company has only a finite amount of resources, and _can't_ support umpteen different versions/dialects ("So, would you people _please_ make up your minds, and as quickly as possible, too . . .")

The entrepreneurs tended to be people working in two to ten programmer shops (some within _much_ larger organizations) who were most concerned with getting working compilers (and libraries) out the door. Questions (and suggestions) about the semantics of the language and implementation details were the meat of their comments.

In regard to the form of the presentations, I found (as I have, all too often, in the past) that the presentations by academics were less than sparkling -- delivered in a monotone, consisting of reading transparencies word-for-word, and boooring. This despite the interesting and wide-ranging nature of what they were talking about.

Representative from medium-sized companies seemed to have better speaking technique, so long as they were talking about the "official" aspects of their products, but too often got maddeningly vague about other details.

What the "hackers" may have lacked in polish, they more than made up in enthusiasm and energy. An outstanding example was J. Savvit, who is almost single-handedly doing a Modula 2 implementation for the IBM 370 architecture (and is one of the very few people in my experience who didn't work for IBM to be enthusiastic about the 370).

Nobody, but nobody, likes any of the existing libraries, and everybody wants one which includes _________ { For I := 0 to Sqr(Maxint) do New(_________); FillIn(_________); End;} -- to lapse into Pascal for a moment. R. Bush loudly and repeatedly disassociated himself from the draft Standard Library, as printed in MODUS #1 ("Its' not _MY_ library, dammit!")

At the end of the day on the 6th, the overall consensus seemed to be that, while not much had been _done_ or _determined_, a lot had been learned, and that all concerned were more than willing to work together.

CONCERNS OF A PROGRAMMER

by Dennis R. Cohen

Before I go into my concerns regarding Modula-2, let me stress that of all the programming languages I use on a somewhat regular basis, Modula-2 is my favorite. The list includes Pascal, C, FORTRAN, Ada, Assembler, and BASIC as well as others that do not get used so frequently.

I attended the recent Stride Faire in Reno, Nevada, where a great number of the presentations were Modula-related and I was both heartened by the interest in the language and distressed by some of the statements made by people who should know better.

My concerns are as follows:

* The lack of a standard library:
Every language that has made a success of itself has been able to guarantee that certain functions will be consistent across all implementations -- FORTRAN code is portable, C programmers depend upon Kernighan & Ritchie conformance, and COBOL programmers have it just as well. This issue MUST be resolved early in the game -- before the onslaught of vastly divergent implementations. There is some effort being made in this regard; however, it is vital that it be done well and quickly.

* Numeric Data Types:
It is necessary that LongReal (or Double), LongInt, and LongCard (or WIDE) be made part of the language definition. If there is not a guarantee of support for multi-precision numeric types at least to this level, I do not believe that Modula-2 will ever make significant inroads into scientific fields--in fact, I would not be able to use it for database development, since numeric data types of high-precision are used regularly in this field. The areas of application are many.

* Support for existing libraries written in other languages:
This is probably one of my major concerns. My programming background is fairly diverse -- databases, word processing, utilities, compilers, operating systems, accounting, and scientific programming. In many of these areas, access to existing libraries of routines saved me from writing large quantities of code. There needs to be some method, even if only the writing of a dummy DEFINITION MODULE, allowing access to these libraries. At Reno, Prof. Wirth stated, "You should rewrite the libraries in Modula-2, you will make them better." This is an interesting pedagogic argument, but it will not happen in the real world. Development companies will not expend the resources to convert existing, functional, and reliable libraries to a new language just so that their development may be done in that new language -- they will do what they have always done, they will choose a language that gives them that access, whether it is FORTRAN, C, Ada, or something else. Yes, even the DOD realized the necessity of providing for existing libraries and required access in the language definition.

I/O is a major concern of mine. This must be standardized in the library and quickly -- great divergence between implementors already exists and before you know it, no program will port between implementations.

When describing numeric data types, it is sufficient to state that the three types above exist in the language and that their ranges either are the same as the base types or properly contain the base ranges -- similar to how the types double and long are defined in C.

Access to existing libraries must be guaranteed and a standard language construct must be defined to signify that access.

A final point is the multi-dimensional open array parameter. While I grant that many programs will never make use of this capability, whole classes of application require it and the people who write these programs also write programs which do not require it and they will not choose to be multilingual in their work where it is unnecessary. In other words, they won't bother to learn Modula-2 when it will not help them do their work.

I want to use Modula-2 in my work, but I cannot start a major development effort in Modula-2 until I have some assurance this basic functionality will be performed and that there will be a consistency between implementations. I do not particularly like C as a language, but I use it on a regular basis because it provides that functionality and consistency and if Modula-2 is to ever achieve "real world" usability, it will have to emulate C to at least that extent.

[Editorial note: (I warned you about leaving blank space)

> Among many other good deeds, Dennis provided this copy more than half a year ago. I lost it three times, and forgot it twice. Finally, you have a chance to see Dennis' concerns. They have not, I believe, suffered any loss of relevance in the months that have passed.
>
> The issue of a Standard Library is the one I worry about first. You may recall that the Ad Hoc Implementors Proposed Standard Library took up a large part of Issue # 1 of this journal. In an excess of zeal, I labeled it with just the words "Modula-2 Standard Library". Unfortunately, this has confused many people into thinking that it has some official status. Not only is this in error, but none of the team who created it is entirely happy with the details of that draft now that more time and experience have revealed a variety of defects and inadequacies in the original design.

rhk]

A Few Modifications to a Modula Standard Library Proposal

Robert J. Nagler

Jeremy A. Siegel

ABSTRACT

After implementing and using the proposed Modula Standard Library as presented in Modula-2 News Issue #1, significant experience and insight was gained about its shortcomings. In this paper, those problems are discussed and a new definition[1] of the library with proposed changes is presented. The modifications presented attempt to improve implementability and usability via a simplified, more modular approach.

"The nice thing about standards is that there are so many different ones to choose from."

Introduction

Standards are a way of life for some and for others they just get in the way. Developing standards is usually quite difficult especially by committee. The Modula Standard Library Ad Hoc Working Group is a committee of exceptional people. These people have worked very hard at the design for the current proposal and have done quite well as a result. However, it is not perfect (nor is any other proposal or standard I have seen). Some of its flaws may appear to be major when in fact, they are superficial in comparison with the problems in other libraries. Therefore, the modified proposal being presented here should not be looked at as a new creation, but as a rework of some ideas passed over or left out by the Working Group.

The discussions presented are often substantiated through examples in other languages without extensive descriptions of those languages. The main languages used are: Ada, C, and Mesa. Pascal is not used as an example, because we feel it has nothing to offer except its simplicity (which is reflected in SimpleIO). We also assume a fair amount of familiarity with the current proposal (keeping a copy at one's side while reading this paper may prove useful).

File Objects

Throughout the history of libraries, the most common classes of file IO supported are binary and text. Binary IO is sometimes called *low level* or *direct* and treats files as arrays of records. The records are sometimes variable in length and others are fixed in size. Text IO on the other hand is a higher level concept. Each record is treated as a line of characters. The file may contain pages, that is, a way of organizing groups of lines.

A sampling of today's structured language libraries reflects the distinction between the two types of file objects: text and binary. Ada has its DIRECT_IO.FILE_TYPE for binary files and a TEXT_IO.FILE_TYPE for texts. C uses its int type as a file handle for its low level routines and

[1] This paper covers conceptual issues used in the new proposal and details are only introduced where necessary. The exact specification of our proposal could not be presented in its entirety due to the limited space provisions. In this paper we do speak of modules, types, and procedures, but their definition should be self-evident from the context in which they are referenced. The Editor has been kind enough to provide interested parties with copies of our complete specification for the cost of copying.

provides a FILE object for text IO. Mesa provides a FileHandle for direct disk IO and StreamHandles are used for texts.

In one of the drafts of the library proposal, text and binary objects were treated separately. The latest draft, however, merges these two objects into one syntactic entity called Files.File. The justification is that the generic procedures for both objects are sufficient in number to warrant this change. There are twelve procedures in the module Files; but upon careful study it becomes apparent that there are not twelve generic operations required.

First, three of the routines in Files may be eliminated since they duplicate functionality available elsewhere: Reset can be accomplished in two procedure calls to FilePositions. Rewrite is simply GetFileName followed by Create. Remove can be implemented with three calls (one of which is Directory.Delete). By some simplification, we have reduced the number of generics from twelve to nine.

One can question the substance of the term "generic". After implementing these procedures on several machines, we observed that the generic interface did not reflect the non-generic implementation. For example, VAX/VMS binary and text files are handled quite differently, so the implementation of Files consists of two parts: binary and text. If an implementation does in fact find it convenient to treat these files the same, there is nothing preventing the implementation from having a shared GenericFiles module which is hidden from the programmer. This type of sharing is considerably more modular than the current design.

Taking a wider view of the library, the modules Text and Binary may be considered extensions to the Files module; they supply procedures which operate on the File object. These procedures are clearly not generic (which is why they were separated) but since they operate on the same syntactic object run-time checks are required to detect inappropriate usage.

For these reasons, we have eliminated the module Files. We eliminated some duplicated functionality in the module Binary and added the necessary functions from Files to create a new module called BinaryIO. Similarly, we created the new module TextIO from the module Text. Each of these new modules exports a type called File syntactically separating the two types of files.

File states, EOL, EOF, success, ...

Any file operation may fail as the result of conditions external to a module's control. Hence, traditional file system interfaces provide facilities to detect failure on most (if not all) entry points. The Ad Hoc Committee's proposal follows this tradition. However, the approach taken is unnecessarily complex and not sufficient to satisfy the needs of robust programs.

In order to simplify the model, we need to address the categories of errors. The Ad Hoc Proposal presents a straightline model for file errors, that is, all errors are defined in one enumeration. Although it is true that one can list all possible error conditions as one type, it is not useful to bind mutually exclusive classes together. For example, one cannot get an existingFile error from a read operation. Similarly, a call to Open would never return outsideFile. This problem was apparent enough to the Ad Hoc Committee that the description of the FileState attempts to categorize the errors into groups based on the routines which can return sub-categories of errors. The problem is the same as presented earlier with merged file types: semantic conventions should be enforced with syntactic definitions.

The error categories we use are defined by two classes of operations: directory transactions and file manipulations. The former concerns manipulation/mapping of external file object names. The routines in Directory are all of this category, as are the procedures which map Modula-2 file objects to external file objects (Create and Open). File manipulations act upon the file object (reading, writing, determining state, ...). These two classifications syntactically enforce the semantic description already imposed by the Ad Hoc Committee's Library. For further discussion, the names of these error lists are AccessResults and IOErrors, respectively.

AccessResults are sufficient to determine the failure or success all directory operations, but IOErrors do not adequately describe the result of all file manipulations. Read operations can fail for two reasons (as defined by both the Ad Hoc Library and our modified version): IOErrors and change in EOF (EOL for text files) state. We consider errors to be abnormal conditions and encountering EOF (instead of reading data) is a normal transaction. This separation imposes the model of EOF (EOL) being treated as out of band data which is read in a system/character set independent fashion. The authors agree with this model[2], but disagree with its implementation.

The difficulty one finds with the Ad Hoc Committee's model can best be demonstrated by an example. The following code fragment is the standard way one reads and processes strings from a text file:

```
LOOP
    Text.ReadString( input, string, state );
    IF Files.EOF( input ) THEN
        EXIT;
    END;

    IF Files.EOL( input ) THEN
        Text.ReadLn( input, state );
        IF state # Files.ok THEN
            Abort( input, "error on readln" );
        END;

    ELSE
        (* Data is okay *)
        Process( string );
    END; (* IF *)
END; (* LOOP *)

IF state # Files.ok THEN
    Abort( input, "error on read" );
END;
```

One simple observation is that ReadString and Process are separated by far too many statements. There is no information processing in the middle of the loop; it is all bookkeeping. Next one observes the occurrance of unnecessary operations. For example, ReadString actually encounters the end-of-line (indicated by EOL being true), but one cannot perform another ReadString without "reading" the EOL that was encountered. Also, the abort condition for "read" is separated from its occurrance. In general, one cannot determine the result of an IO operation based on a single returned value.

To resolve this problem, we enumerate the various checks associated with a file operation as follows:

```
TYPE
    FileStates = ( ok, endOfLine[3], endOfFile, error );
```

The constants indicate the reason for the result of the previous file operation. If the state of the file is ok, then the operation was successful. An error state indicates that one of the IOErrors occurred and the program can call a procedure to determine the exact nature of the failure. We address the robust programming issue with this state as well: a program will be terminated by the file modules, if and only if the file being operated upon is in an error state. Therefore, all terminations are detectable whereas in the Ad Hoc Library a program may be terminated for reasons which are not detectable (e.g. attempted Text operation on a file in binMode).

Encountering endOfFile and endOfLine indicates the previous read operation failed for that reason. The endOfFile condition is sticky in that an ensuing read operation will fail as a result of IOErrors.readPastEOF (unless the file position is changed). The endOfLine is treated as out of band data and signifies that no data was read on the last read operation. However, one does not have to call

[2] The discussion of EOF (EOL) as procedural versus data driven versus exceptions, etc. is not part of the scope of this paper. The authors believe that our model is simple enough that it can be implemented on any system that treats text files as sequence of lines of characters. Clearly, one could use a more complicated mechanism which would improve efficiency (exceptions come to mind), but such systems are not simple to implement and have definite portability problems given the definition of Modula-2 at this time.

[3] We are using the FileStates from our module TextIO. The module BinaryIO has an identical enumeration excluding the endOfLine constant.

ReadLn to clear this condition. Ensuing reads are permitted and behave as one would expect (they start reading after the end-of-line that was just encountered).

To better appreciate our model, we will rewrite the previous example in terms of our proposal.

```
LOOP
   TextIO.ReadString( input, string, state );
   CASE state OF
      TextIO.ok:
         Process( String );
   |
      TextIO.endOfLine: (* nothing to process *)
   |
      TextIO.endOfFile:
         EXIT;
   |
      TextIO.error:
         Abort( input, "error on read" );
   END; (* CASE *)
END; (* LOOP *)
```

StandardIO

The concept of default input and output is in many systems (Ada, C/UNIX, Fortran, and Pascal to name a few). One needs such a mechanism to initiate file IO with the outside world in a portable program (given that we assume file names are non-portable). Given this history and logic, the Ad Hoc Committee defined the module SimpleIO. Most libraries provide facilities for changing the external file object associated with the default files. Hence, the need for StandardIO in the Ad Hoc Library. These features are all very useful; it is the extensions to the traditional model that cause problems.

One problematic extension is echoing standard input to standard output. First, the authors know of no library which implements echoing in this fashion. The concept of echoing at the file system level (versus the device driver level) introduces many interesting questions. What happens when one executes an UndoRead followed by another Read? Is the character echoed again? Is EOL echoed when EOL becomes TRUE or only when ReadLn is called? For files which are implemented as interactive devices: if the device is running in local echo mode (most commonly associated with half-duplex terminals), how does SetEchoMode(noEcho) cause the local echoing to be terminated? For line buffered devices which have interactive line editing: if part of the line is read with echoing off after which echoing is turned on, do ensuing reads cause the characters already buffered to be echoed? For devices that already echo: does the library cause duplicate echoes?

We could list other problems associated with this mechanism, but it would be in vain. This sequence of questions is sufficient to demonstrate the problems associated with echoing at the file system level and justifies its elimination from StandardIO.

Logging standard output is another feature that is not available in other libraries, but is provided in the Ad Hoc Proposal. Granted that this feature may be useful under certain circumstances, it is not clear that its usefulness outweighs its implementation overhead.

Text Manipulations

The Ad Hoc Library allows all modes of access on text files; more importantly, text files can be opened in readWrite mode. Allowing reads and writes from the same text file can be extremely complicated and difficult to implement on some systems. It is ill defined under certain circumstances. For example, if one opens a text file for reading and writing, positions[4] the file pointer to the middle of a line, and the string written extends beyond the end of the current line in the file, does the line get extended or is a line break lost? In fact, many libraries (C/UNIX, Mesa, Ada, Pascal) treat text files as one way

[4] Note that file positioning is only allowed on files with read access.

streams, that is, read only or write only. For these reasons, we have not included read/write text files in our library.

Our new proposal does not include the ability to calculate file indices for text files. The Ad Hoc Library noted that this facility was not portable and results were unpredictable. Calculating file positions on text files can lead to ambiguities similar to those of read/write text files described above.

The Ad Hoc Library included a conditional read facility (CondRead) for text files. Its purpose was to provide non-blocking IO for text files. Clearly, this utility is only applicable to interactive devices implemented as files. We believe this contradicts a goal of the library specified in Modus Issue #1: "Do not specify the operating system ...".

Other Modules

The status of the module Terminal is the most questionable of all the modules. One needs it to perform interactive IO with the user, therefore it is crucial to interactive programs. At the same time, interactive IO is not portable and presents the most problems. For example, does ReadString do line editing? (What is line editing?) If a capability does not exist, what happens? We have included a definition in our library that attempts to solve the problems associated with interactive IO. However, this module is quite complex which leads one to believe that the solution may be better left to an implementation dependent library. We are still studying the issues involved.

The Ad Hoc Library's Storage.CondAllocate is sufficient to implement a program robust against memory exhaustion, but it is not simple to use. One would like to use NEW with Storage.CondAllocate, because it gives that extra level of protection against allocating objects of the incorrect size. NEW, however, requires a procedure named ALLOCATE with two parameters whereas CondAllocate has three parameters. Instead of CondAllocate, we introduce a new module called CondStorage which exports two routines that match the requirements of NEW and DISPOSE. The definition of Storage is such that it is defined to be a higher level layer on top of CondStorage so as not to introduce dual memory managers.

The module Program has been eliminated in our proposal. The execution of sub-programs is intimately connected with the underlying system and thus cannot be portably defined. We list some questions to better state our case. Can non-Modula-2 programs be executed? If module A is imported by module B and module B executes sub-program C which also imports A, can data be shared between modules B and C via A? What is the programName passed to Program.Call: a file name, module name, or something else? Why are SetInitialization and SetTermination used in systems which do not share data between programs? How does the user get a hold of the reason (CallResult) passed to Program.Terminate?

Conclusion

The authors would like to thank Jeanette Symons for her patience while providing continuing support and critiques of the new library proposal. We appreciate the input from Jon Bondy, Chris Jewell, Ted Powell, and Roger Sumner based on their experience implementing, using, and further defining the Ad Hoc Committee's Proposal. Finally, we are grateful for the insight into the Ad Hoc Committee's decisions provided by Randy Bush.

This paper has served to introduce the reader to what the authors believe are the major drawbacks of the Ad Hoc Committee's Proposed Standard Library. The complete specification of our new proposal addresses issues similar to these but on a smaller scale. The fundamental issues we have with the Ad Hoc proposal are: lack of modularity, lack of specification, and excessive numbers of features. Our new proposal addresses these issues with the scope of the Committee's goals. We believe the modifications to the Ad Hoc Committee's Library proposed in this paper will greatly increase its usefulness and implementability.

Proposal For a Standard Library and an Extension to Modula-2

By
Martin Odersky, Peter Sollich & Mike Weisert
Borland International
4585 Scotts Valley Drive
Scotts Valley, CA 95066

Introduction

Many proposals have been made for a set of Standard Library modules, however, they all have come up against the same problem of what to do in error situations. This problem is (unfortunately) caused by the definition of Modula-2. This is why we propose a modification to the language and at the same time we present a clear, and usable Standard Library which results from a simple but well thought-out extension to Modula-2.

The problems which other library proposals have come to face is how to deal with error checking across module boundaries. The inevitable solution is to use a success/failure flag that can be tested after each call to a library procedure. This is undesirable in either of the two methods proposed.

The first method sets a flag after each successful or unsuccessful operation. This means that the user program must check the value of the flag after each library call to ensure it is not changed by a subsequent call. The second method is to initialize the flag to a successful value and then only set the flag to a failure status if something goes wrong. While this saves the user program from continually checking the error flag, it makes it very difficult for the program to determine where and when an error occurred.

Another method which has been proposed is to allow the user program to install an error handling procedure which is called when the library detects an error condition. This may appear to solve the problem of constantly checking flags, but it does not. This is because an error procedure can only do one of two things. It can either print an error message and then terminate the program or it can try to recover (possibly set a flag) and then return to the place where the error occurred. Thus to recover from an error condition without termination, the error handler must set some flag which can be checked by the calling program. This method does not solve the problem.

A variation of the above methods has been proposed which allows the user program to set a error handling mode. This mode is then checked by the library to determine whether the user will be checking the error flags or if the library procedures should terminate on any error. This method keeps simple programs simple, but presents many problem to the designer of a general purpose library which is intended to be robust. The major problem with this method is that several processes may be setting the library mode to different values, thus causing unpredictable results.

Since the problem does not seem to have been solved by any previous proposals, we have thought hard about any possible solution. Our conclusion is that elegant error handling can not be achieved in the Modula-2 language as it stands. We propose an extension to the language which makes error handling efficient and effective. The extension we propose is based on the error handling method used in the language Ada, called EXCEPTIONs.

Briefly, exceptions allow the user program to trap error conditions which are generated in some library module. Since the library module can not know what the user wants to do in an error situation, it simply raises an exception which the user can either handle or ignore. Program control never returns to the point where the error occurred and so no status flags need to be set and checked. Thus simple programs remain simple, while larger modules have the flexibility and power to handle error conditions easily.

This has a profound effect on our proposed library modules. Instead of exporting some status flag (or having extra status parameters) the library exports exceptions. The user modules can choose to handle exceptions or not. If not, the Modula-2 system will trap the exception and issue an appropriate error message. This greatly simplifies library modules. The following presents explanation and justification of our proposed library modules.

Proposals for Library Modules

The input/output library

When dealing with I/O, error handling is very important. Disks get full, are write-protected or have bad sectors; printers are not on line. Disk files opened for reading may not be present due to improper names, wrong drive codes, etc.

Another concern is reliability. We have to make sure clients of I/O modules cannot disturb the function of the modules.

Most important to users is, of course, convenience. This is especially true for the I/O library, as it is going to be used very often. Common operations should be expressed as simple as possible. Procedures should use a minimum number of parameters. Users also don't want too many library modules.

There seem to be two fundamental modes of I/O: there is textual I/O (in Pascal performed on variables of type TEXT), and there is binary I/O (involving variables of type FILE OF <type> in Pascal). Textual I/O involves conversions and is usually not restricted to disks, but is also possible on terminals, printers, and other sequential devices.

We therefore decided to use a module called "Texts" for textual I/O and another module, "Files", for binary I/O.

PAGE 14

The Module Texts

We will first look at "Texts", as one naturally uses this one first. Some of the things to notice in the definition module are as follows:

Type TEXT was declared as a subrange [1..16]. This is done to allow user written I/O driver procedures to be installed (via ConnectDriver). The following demonstrates that TEXT procedures can be used in an extensible manner. A user could define a low-level driver for any device. Imagine for example a window handler. If we use

```
  TYPE window = TEXT;
```

then the user module WindowHandler can have its private tables (arrays indexed by TEXT variables) containing the properties of the windows. We have only to implement a simple driver procedure for writing single characters, install it via ConnectDriver and then we can use any of the formatted write procedures to write on our windows.

REAL and LONGINT I/O are included in this module rather than a separate one. However, LONGREAL I/O is packaged separately.

Done(t) denotes successful conversion on numeric input, rather than denoting completion of a procedure. This is because other errors that occur during library calls are trapped by the exception facility. Exceptions are raised for illegal operations or when Texts reaches its limits.

OpenText returns a boolean result indicating whether the text was found. It has the side-effect of opening the Text if it was found. A boolean result makes searching for texts and opening them very elegant. One may write, for example

```
  REPEAT
    WriteString("Filename ?>"); ReadString(filename);
  UNTIL (filename = "") OR OpenText(text,filename);
  IF filename # "" THEN
    (* process the file *)
  END
```

There is the opinion, however, that one should not use functions with side effects. We opt for the more elegant solution. A proposed standard for library modules (Modula-2 News, Issue # 1, January 1985) would require the following:

```
REPEAT
  WriteString("Filename ?>"); ReadString(filename);
  IF filename # "" THEN
    Open(file,filename,textMode,readOnly,state)
  END;
 UNTIL (filename = "") OR (state = ok);
 IF filename # "" THEN
   (* process the file *)
 END
```

We think users will like our version of Open better. It has just two parameters instead of five. The other version is usable, but it takes more effort to remember the correct number and sequence of parameters, to think of declaring the additional state variable and to get the logic correct.

TextFile returns the file connected to a text. This is useful for doing low-level (e.g. SetPos) operations on texts. See discussion of module Files below.

The variable haltOnControlC determines how programs react to ^S and ^C during a call to a TEXT procedure. This is a CP/M specific feature and is not proposed as standard.

In Texts, exceptions are only used for errors that should normally not happen. It usually makes little sense to catch these exceptions. However, the module Texts has to rely upon the module Files for disk I/O. If the Disk gets full, Files would raise the exception DiskFull, which propagates through Texts to the user program. To catch this exception, one has to import this exception from Files, even if nothing else is imported from Files.

The definition module is as follows:

```
DEFINITION MODULE Texts;
  FROM Files IMPORT FILE;

  TYPE TEXT = [1..16];

  VAR input,output,console: TEXT;

  PROCEDURE ReadChar    (t: TEXT; VAR ch: CHAR);
  PROCEDURE ReadString  (t: TEXT; VAR s: ARRAY OF CHAR);
  PROCEDURE ReadInt     (t: TEXT; VAR i: INTEGER);
  PROCEDURE ReadCard    (t: TEXT; VAR c: CARDINAL);
  PROCEDURE ReadLong    (t: TEXT; VAR l: LONGINT);
  PROCEDURE ReadReal    (t: TEXT; VAR r: REAL);
  PROCEDURE ReadLn      (t: TEXT);
  PROCEDURE ReadLine    (t: TEXT; VAR s: ARRAY OF CHAR);

  PROCEDURE WriteChar   (t: TEXT; ch: CHAR);
  PROCEDURE WriteString(t: TEXT; s: ARRAY OF CHAR);
  PROCEDURE WriteInt    (t: TEXT; i: INTEGER;  n: CARDINAL);
  PROCEDURE WriteCard   (t: TEXT; c: CARDINAL; n: CARDINAL);
  PROCEDURE WriteLong   (t: TEXT; l: LONGINT;  n: CARDINAL);
  PROCEDURE WriteReal   (t: TEXT; r: REAL;     n: CARDINAL;
                                          digits: INTEGER);
  PROCEDURE WriteLn     (t: TEXT);
```

PAGE 16

```
  PROCEDURE ReadAgain  (t: TEXT);
  PROCEDURE Done       (t: TEXT): BOOLEAN;
  PROCEDURE EOLN       (t: TEXT): BOOLEAN;
  PROCEDURE EOT        (t: TEXT): BOOLEAN;
  PROCEDURE Col        (t: TEXT): CARDINAL;
  PROCEDURE SetCol     (t: TEXT; column: CARDINAL);
  PROCEDURE TextFile   (t: TEXT): FILE;

  PROCEDURE OpenText   (VAR t: TEXT; name: ARRAY OF CHAR): BOOLEAN;
  PROCEDURE CreateText (VAR t: TEXT; name: ARRAY OF CHAR);
  PROCEDURE CloseText  (VAR t: TEXT);

  CONST EOL=36C;

  TYPE TextDriver = PROCEDURE(TEXT, VAR CHAR);
  PROCEDURE ConnectDriver(VAR t: TEXT; p: TextDriver);

  VAR haltOnControlC: BOOLEAN;     (* CP/M specific *)

  EXCEPTION TextNotOpen, TooManyTexts;
END Texts.
```

The Module Files

The Module Files is meant to perform binary I/O on disks and similar devices. When examining the module note the following:

Type FILE is an opaque type (in effect a pointer to a file descriptor). Like OpenText, Open returns a boolean result. Flush empties the file buffer to detect DiskFull errors at once.

Using file variables for Delete and Rename was done because most applications which delete and rename files deal with files that are already open. Many programs generate temporary files which are then either deleted or renamed. Note that Delete and Rename have the side-effect of closing the file.

Being able to obtain the name of a file is useful when a file variable is passed to another procedure or module (e.g. to issue meaningful error messages).

The alternative to using LONGINT for file positions would have been to use some record type. The advantage of LONGINT is that we can easily do calculations on it and that it can be returned by functions. If we want to advance by, say, one kilobyte, we can simply write:

```
  SetPos(f,NextPos(f)+1024L)     or     SetPos(f,FileSize(f))
```

to go to the end of the file. Notice the use of NextPos() to obtain the current position in the file.

The meaning of the read and write procedures is obvious. The functions ReadBytes returns the number of bytes actually read. The two procedures, NoTrailer and ResetSys, were developed exclusively for the CP/M environment. They are proposed only for systems with similar problems.

The meanings of the exceptions used in Files are as follows:

- EndError   means we have attempted to read or position past the end of file.
- StatusError   is raised for operations other than Open or Create on files that are not open.
- UseError   means access to the file is impossible, because it is write protected, etc.
- DeviceError   is raised for hardware failures.
- DiskFull   means writing to the disk is impossible.

The definition module of Files looks as follows:

```
DEFINITION MODULE Files;
  FROM SYSTEM IMPORT BYTE, WORD, ADDRESS;

  TYPE FILE;

  PROCEDURE Open      (VAR f: FILE; name: ARRAY OF CHAR): BOOLEAN;
  PROCEDURE Create    (VAR f: FILE; name: ARRAY OF CHAR);
  PROCEDURE Close     (VAR f: FILE);
  PROCEDURE Delete    (VAR f: FILE);
  PROCEDURE Rename    (VAR f: FILE; name: ARRAY OF CHAR);

  PROCEDURE GetName   (f: FILE; VAR name: ARRAY OF CHAR);
  PROCEDURE FileSize  (f: FILE): LONGINT;
  PROCEDURE EOF       (f: FILE): BOOLEAN;

  PROCEDURE ReadByte  (f: FILE; VAR ch: BYTE);
  PROCEDURE ReadWord  (f: FILE; VAR w: WORD);
  PROCEDURE ReadRec   (f: FILE; VAR rec: ARRAY OF WORD);
  PROCEDURE ReadBytes (f: FILE; buf: ADDRESS; nbytes: CARDINAL): CARDINAL;

  PROCEDURE WriteByte (f: FILE; ch: BYTE);
  PROCEDURE WriteWord (f: FILE; w: WORD);
  PROCEDURE WriteRec  (f: FILE; VAR rec: ARRAY OF WORD);
  PROCEDURE WriteBytes(f: FILE; buf: ADDRESS; nbytes: CARDINAL);

  PROCEDURE Flush     (f: FILE);
  PROCEDURE NextPos   (f: FILE): LONGINT;
  PROCEDURE SetPos    (f: FILE; pos: LONGINT);

  PROCEDURE NoTrailer (f: FILE);      (* CP/M specific *)
  PROCEDURE ResetSys  ();             (* procedures.   *)

  EXCEPTION EndError, StatusError, UseError, DeviceError, DiskFull;
END Files.
```

<u>Other I/O Modules:</u>

The module InOut is just there because Wirth's book says it is a standard module. We ourselves use it very rarely.

```
DEFINITION MODULE InOut;
  CONST EOL=36C;
  VAR Done: BOOLEAN;
```

PAGE 18

```
     termCH: CHAR;

   PROCEDURE OpenInput(defext: ARRAY OF CHAR);
   PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
   PROCEDURE CloseInput;
   PROCEDURE CloseOutput;

   PROCEDURE Read(VAR ch: CHAR);
   PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
   PROCEDURE ReadInt(VAR x: INTEGER);
   PROCEDURE ReadCard(VAR x: CARDINAL);

   PROCEDURE Write(ch: CHAR);
   PROCEDURE WriteLn;
   PROCEDURE WriteString(s: ARRAY OF CHAR);
   PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
   PROCEDURE WriteCard(x,n: CARDINAL);
   PROCEDURE WriteHex(x,n: CARDINAL);
   PROCEDURE WriteOct(x,n: CARDINAL);

   PROCEDURE ReadReal(VAR x: REAL);
   PROCEDURE WriteReal(x: REAL; n,digits: CARDINAL);

END InOut.
```

The module Terminal gives low level access to the console:

```
DEFINITION MODULE Terminal;

   PROCEDURE ReadChar(VAR ch: CHAR);
   PROCEDURE BusyRead(VAR ch: CHAR);
   PROCEDURE ReadAgain;
   PROCEDURE ReadLine(VAR s: ARRAY OF CHAR);

   PROCEDURE WriteChar(ch: CHAR);
   PROCEDURE WriteLn;
   PROCEDURE WriteString(s: ARRAY OF CHAR);

   VAR numRows,numCols: CARDINAL;

   PROCEDURE ClearScreen;
   PROCEDURE GotoXY(col,row: CARDINAL);

   PROCEDURE InitScreen;
   PROCEDURE ExitScreen;

   TYPE SpecialOps = (clearEol,insertDelete,highlightNormal);
        OpSet      = SET OF SpecialOps;

   VAR  available:   OpSet;

   PROCEDURE ClearToEOL;
   PROCEDURE InsertLine;
   PROCEDURE DeleteLine;

   PROCEDURE Highlight;
   PROCEDURE Normal;
END Terminal.
```

PAGE 19

Clearing the screen and positioning the cursor are assumed to be present on any system. *InitScreen* and *ExitScreen* provide for terminals that must be initialized on entry or reset on exit. The set variable *available* determines whether clearing to the end of the line, inserting or deleting lines and highlighting work on the specific installation.

The Module Comline

The module Comline provides for commandline arguments to programs and input/output redirection.

```
DEFINITION MODULE ComLine;
  FROM Texts IMPORT TEXT;

  VAR commandLine    : TEXT;
      inName,outName: ARRAY [0..19] OF CHAR;
      progName       : ARRAY [0..7] OF CHAR;

  PROCEDURE RedirectInput;
  PROCEDURE RedirectOutput;

  PROCEDURE PromptFor(prompt: ARRAY OF CHAR; VAR s: ARRAY OF CHAR);
END ComLine.
```

The commandline is declared as a text, so we can read anything out of it. The variables *inName* and *outName* contain the names of the redirection arguments, *progName* contains the name of the program. *RedirectInput* and *RedirectOutput* enable I/O redirection. The procedure *PromptFor* reads *s* from the commandline, if it is not yet exhausted. Otherwise it writes the prompt on the screen and reads *s* from the terminal. In this way arguments to programs can be given on the commandline or via an explicit prompt.

The Module Convert

The module Convert provides for conversions from strings to numbers and vice versa. As in Texts, we did not separate REAL conversions from the rest.

```
DEFINITION MODULE Convert;

  PROCEDURE StrToInt (VAR s:ARRAY OF CHAR; VAR i:INTEGER)  :BOOLEAN;
  PROCEDURE StrToCard(VAR s:ARRAY OF CHAR; VAR c:CARDINAL):BOOLEAN;
  PROCEDURE StrToLong(VAR s:ARRAY OF CHAR; VAR l:LONGINT)  :BOOLEAN;
  PROCEDURE StrToReal(VAR s:ARRAY OF CHAR; VAR r:REAL)     :BOOLEAN;

  PROCEDURE IntToStr (i: INTEGER;  VAR s: ARRAY OF CHAR);
  PROCEDURE CardToStr(c: CARDINAL; VAR s: ARRAY OF CHAR);
  PROCEDURE LongToStr(l: LONGINT;  VAR s: ARRAY OF CHAR);
  PROCEDURE RealToStr(r: REAL;     VAR s: ARRAY OF CHAR;
                                          digits: INTEGER);
  EXCEPTION TooLarge;
END Convert.
```

Those procedures that convert from strings to numbers return an explicit boolean result to indicate success of the conversion. The conversions from numbers to strings can only fail because of too little space in the result string. For that case, the exception TooLarge is raised.

The philosophy is that incorrect syntax in numbers is to be expected quite often, so the programmer should check for it. The result string being too small is usually a programming error.

Other modules

The Module Storage

```
DEFINITION MODULE STORAGE;
  FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE ALLOCATE  (VAR a: ADDRESS; size: CARDINAL);
  PROCEDURE DEALLOCATE(VAR a: ADDRESS; size: CARDINAL);

  PROCEDURE MARK      (VAR a: ADDRESS);
  PROCEDURE RELEASE   (VAR a: ADDRESS);

  PROCEDURE FREEMEM   (): CARDINAL;
END STORAGE.
```

The procedure FREEMEM returns the largest block that can be allocated, i.e. the largest gap in the free list. No problem exists with intervening interrupts, because each process uses a separate heap.

Running out of memory causes the exception OUTOFMEMORY (exported by SYSTEM) to be raised. This may be caused either by a call to ALLOCATE or a procedure call that causes the runtime stack to meet the heap.

The Module Strings

Strings performs several quite useful functions. The reason for featuring Append instead of the more familiar Concat is that almost all Concat's are in fact more easily expressed as Append's.

```
DEFINITION MODULE Strings;

  TYPE String = ARRAY[0..80] OF CHAR;

  PROCEDURE Length (VAR str: ARRAY OF CHAR): CARDINAL;
  PROCEDURE Pos    (substr,str: ARRAY OF CHAR): CARDINAL;

  PROCEDURE Insert (substr: ARRAY OF CHAR;
                    VAR str: ARRAY OF CHAR; inx: CARDINAL);
  PROCEDURE Delete (VAR str: ARRAY OF CHAR; inx,len: CARDINAL);

  PROCEDURE Append (substr: ARRAY OF CHAR;
                    VAR str: ARRAY OF CHAR);
  PROCEDURE Copy    (VAR str: ARRAY OF CHAR; inx,len: CARDINAL;
                     VAR result: ARRAY OF CHAR);

  PROCEDURE CAPS    (VAR str: ARRAY OF CHAR);

  EXCEPTION StringError;
END Strings.
```

The procedure CAPS provides a similar function for strings as CAP is for characters. The exception StringError is raised if the resulting string is too long to fit into the result variable.

The Modules Mathlib and Longmath

These modules provide the usual mathematical functions for REAL and LONGREAL arguments.

```
DEFINITION MODULE MathLib;

  PROCEDURE Sqrt      (x: REAL): REAL;
  PROCEDURE Exp       (x: REAL): REAL;
  PROCEDURE Ln        (x: REAL): REAL;
  PROCEDURE Sin       (x: REAL): REAL;
  PROCEDURE Cos       (x: REAL): REAL;
  PROCEDURE Arctan    (x: REAL): REAL;
  PROCEDURE Entier    (x: REAL): INTEGER;
  PROCEDURE Randomize(n: CARDINAL);
  PROCEDURE Random    (): REAL;

  EXCEPTION ArgumentError;
END MathLib.

DEFINITION MODULE LongMath;

  PROCEDURE Sqrt   (x: LONGREAL): LONGREAL;
  PROCEDURE Exp    (x: LONGREAL): LONGREAL;
  PROCEDURE Ln     (x: LONGREAL): LONGREAL;
  PROCEDURE Sin    (x: LONGREAL): LONGREAL;
  PROCEDURE Cos    (x: LONGREAL): LONGREAL;
  PROCEDURE Arctan(x: LONGREAL): LONGREAL;
  PROCEDURE Entier(x: LONGREAL): LONGINT;

  EXCEPTION ArgumentError;
END LongMath.
```

Both math libraries raise the exception ArgumentError if the function cannot be calculated because arguments are out of range.

The Module Processes

This is just the standard module as given by Wirth in his book.

```
DEFINITION MODULE Processes;

  TYPE SIGNAL;
```

PAGE 22

```
PROCEDURE StartProcess(P: PROC; n: CARDINAL);
      (* start a concurrent process with program P
       * and workspace of size n *)

PROCEDURE SEND(VAR s: SIGNAL);
      (* one process waiting for s is resumed *)

PROCEDURE WAIT(VAR s: SIGNAL);
      (* wait for some other process to send s *)

PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
      (* Awaited(s) =
       *    "at least one process is waiting for s" *)

PROCEDURE Init(VAR s: SIGNAL);
      (* compulsory initialization *)

EXCEPTION DeadLock;

END Processes.
```

For any practical application, Wirth's module will of course not suffice. We take it more as an illustration of how to make use of the coroutine facilities. We have written several process schedulers for different purposes, including (not very demanding) real-time applications. Wirth's proposal has the technical drawback of not avoiding indefinite overtaking.

The Module Loader

This module serves to load overlays via the procedure Call.

```
DEFINITION MODULE Loader;

  PROCEDURE Call(modName: ARRAY OF CHAR);

  EXCEPTION LoadError;
END Loader.
```

The Module Doubles

The module Doubles provides conversions and I/O for the LONGREAL data type. As the conversions are fairly complicated and not used very often, we did not include these procedures in Convert and Texts. The variable legal is used to check for legal input. The exception TooLarge (imported from Convert) is raised by DoubleToStr if the result is too large to fit into the result string.

```
DEFINITION MODULE Doubles;
  FROM Texts IMPORT TEXT;

  VAR legal: BOOLEAN;

  PROCEDURE StrToDouble (VAR s: ARRAY OF CHAR;
                         VAR r: LONGREAL      ) : BOOLEAN;
```

```
PROCEDURE DoubleToStr (     r: LONGREAL;
                        VAR s: ARRAY OF CHAR; digits: INTEGER);

PROCEDURE ReadDouble  (t: TEXT; VAR r: LONGREAL);
PROCEDURE WriteDouble (t: TEXT; r: LONGREAL;
                       n: CARDINAL; digits: INTEGER);

END Doubles.
```

Syntax and Semantics of Proposed Exception Handling

Declaration of Exceptions

An exception declaration is similar to other declarations. There is a special keyword (EXCEPTION) followed by a list of identifiers. Example (taken from the module Files):

```
  EXCEPTION
    EndError, StatusError, UseError, DeviceError, DiskFull;
```

All the usual scope rules of Modula apply. Exception identifiers can be exported and imported like normal Modula identifiers.

Raising Exceptions

Exceptions are raised when the program detects an error condition. (For example, when the module Files has detected that the disk is full). Raising an exception will transfer control to an exception handler, either provided by the user or the system.

A program may raise an exception with the reserved word RAISE followed by the exception identifier and optionally by a string. Example (taken from MathLib):

```
  IF x < 0.0 THEN
    RAISE ArgumentError, 'Negative argument for Sqrt';
  END;
```

When an exception is raised, the system looks in the current procedure for a matching exception handler. If none is found, the calling procedure is examined, then the caller of that procedure, and so on, until a matching exception handler is found. This handler is then executed, and the procedure containing the handler is exited. If no handler is found, the system prints the exception identifier's name and the optional message string.

Handling Exceptions

Exception handlers are written at the end of procedures and modules to handle exceptions issued by a RAISE statement. The syntax is similar to the familiar CASE statement. Example of a save procedure containing an exception handler:

```
PROCEDURE SaveFile;
BEGIN
  (* Code to write something to disk *)
EXCEPTION
  | DiskFull :
      Terminal.WriteString("Disk Full, Press <ESC>");
      REPEAT Terminal.ReadChar(ch) UNTIL ch = CHR(27);
  | DeviceError:
      Terminal.WriteString("Bad Disk, Press <ESC> ");
      REPEAT Terminal.ReadChar(ch) UNTIL ch = CHR(27);
END SaveFile;
```

An exception handler may catch and then pass on an error condition with a special form of the RAISE statement. This form is not followed by an exception identifier or message string.

Conclusion

As exceptions are an extension to Modula, we have to have good reasons for including them.

The three error handling methods used by other implementations are the following:

a) return an explicit success parameter after every operation
b) allow user programs to install error handling procedures
c) set a mode indicating whether the user program wants to check for itself or be aborted on error

Our conclusion is to prefer exceptions to these solutions. One of the advantages is that handlers are present statically in the program text, near the place they are needed. They are much clearer than modes or error handlers that are set somewhere. They also follow the nested structure of the language itself. We think, they fit very nicely within the framework of the language Modula.

If you have read this far, we thank you for your time. We would like to hear any comments, criticisms, or suggestions you may have. Send them to Mike Weisert, Borland International. Phone: (408) 438-8400 x421.

PAGE 25

An Implementation of the Proposed Standard Library for the Unix Operating System

Morris Djavaheri

Djavaheri Bros., 697 Saturn Court, Foster City, California 94044

ABSTRACT

This is a short report about our implementation of the British Standards Institute's ad hoc committee's proposal for a Standard Modula-2 Library. While trying to stay as close to the proposed standard library definition as possible we ran into a number of issues not properly addressed. These issues and our current solutions are discussed.

1. Unix Portability

One of the most basic problems is that there is no standard Unix interface which we could use to build our libraries and remain compatible across the different implementations of Unix. (There is no standard for an interface to an operating system in general. Though one is being developed for Unix.) Therefore we chose to use a set of basic primatives which are always needed, such as: `open`, `read`, `write`, `close`, `fork`, etc. From these basic primitives we implemented an interface between the Unix routines to Modula-2 procedures. These provided essentially the same interface, but they also provide a portability layer between the Modula-2/68 libraries and the different implementations of Unix primitives.

For example, the definition `open(2)` depends on the implementation of Unix. For each Unix implementation it can have a different number of parameters, and these parameters may have different meanings.

Under 4.2 BSD	`open(path,flag,mode)`
Under UNOS	`open(path,flag,mode,Min,Max)`
our internal	`Mopen(path,flag,mode)`

Note that UNOS is a Unix like operating system sold by Charles River Data Systems. 4.2 BSD is the implementation of Unix from UC Berkeley available for a number of MC68000 hardware systems including Integrated Solutions, and Sun Microsystems.

The main purpose for our abstraction of the Unix primitives is to isolate our library from the lack in Unix of a portable interface to the operating system primitives. This makes retargeting the compiler, linker, and libraries to a different operating system or even to a ROM (special applications) much easier to do. In a sense, we have defined our own interface to the host operating system to make the library routines highly portable.

2. Specific Implementation Problems

The Unix operation systems provides a rich file system interface to application programs for manipulating files and hardware devices. With Unix the access to disk files, pipes, terminals and other devices is all very similar. Compared with the Unix programming environment, the proposed library is primitive.

2.1. Filemodes

Every file under Unix has three levels of access: user, group, and world. Each level has three types of permitted access (permissions): read, write and execute. When a file is created on Unix, one needs to have a default permission already set. Also an application may need to change the permission settings for a file after it has been created.

Except for the `ReadWrite` mode set during `Open` there is no place in the proposed library where such a capability is discussed. How can one map the `ReadWrite` mode of the library definition to Unix file permission modes?

Currently our solution is to allow an application to use the `Mmask` library routine to adjust file modes.

2.2. Truncate

The truncation of a file using an operating system primitive is only available with 4.2 BSD

PAGE 26

Unix. This means other implementations of Unix will force Truncate to copy the data to a new and shorter file. This can be an extremely time and a space consuming process.

2.3. RealIO

The proposed library allows RealIO to be done only with files controlled by SimpleIO. This most likely is an oversite. RealIO should be possible with all types of files, especially to files controlled by such modules as Text or Terminal.

2.4. SimpleIO

Simple I/O provides control over the echoing mode of the StdOut file. Under Unix echo is controlled by the device driver software, interfacing the hardware to users application. A majority of terminal device drivers process the input characters based on application controlled settings of device driver characteristics. The proposed library does not provide a general interface for controlling and setting device drivers with this capability. Also it is not clear if user echo means device driver echoing or SimpleIO independent echoing. (see raw and cooked modes on Unix)

2.5. Terminal

This is a very basic interface to an output device. On Unix termcap is a general purpose database used with libraries to provide a terminal independent interface to application program for the controlling of screen I/O applications. Therefore a mapping of Terminal to termcap under Unix O/S is the correct choice, but makes the implementation of the procedure more complicated. The definition Terminal should include some facility for taking advantage of sophisticated hardware and operating system device drivers.

2.6. I/O Buffering

Providing good I/O buffering to applications under Unix is very important. (It is almost always needed for high performance applications.) Therefore we provided buffering for both input and output data files. Though Unix has system internal buffering, performance studies show an application's performance will improve dramaticly with proper buffering of the data files. For example a Modula-2 pretty print program using itself as input runs 4 times faster with buffering than with no buffering The proposed libraries do not provide and interface for buffer I/O. Also having a way to adjust the number of buffers and each buffer's size is important.

2.7. Process Creation and Termination

The standard Modula-2 low level primitives such as NEWPROCESS or TRANSFER are not suitable under Unix for multi-processing. Though these primitives can be implemented within the context of a single Unix process, it is better to use the Unix equivalents of these operations as a starting place for process control primitives. Also the proposed library interfaces Call and Terminate are not adequate for use on Unix, since it expects parameters to be passed to the new process.

2.8. Error Conditions

Most of the proposed library procedures have a rich set of error conditions associted with them, except for process control operations. There are a group of error conditions under Unix which can't be mapped to the Terminate code. Also there is no general interface to lookup an error message string for error codes with the proposed library.

Lastly, error conditions must be checked after each operation. For example:

```
ReadChar(file,ch,state)
IF state <> ok
 THEN
  Message("Can't read from the file")
 END;
```

Generaly application programs do a lot of character reads and writes instead of block reads and writes, this makes it necessary to check the file state after each I/O operation. This can lead to an intolerable performance overhead.

An alternative is to have an exception handling interface like Lisp, or PL/1 to reduce the checking for I/O errors and allow exceptions to be handled with less impact on performance.

3. Summary

The proposed library was implemented with less than 5000 lines of Modula-2. This includes some procedures for error messages, and I/O buffering not defined in the current library proposal. The implementation effort now provides us a library for use in the Unix environment that is likely to be close to the final standard library definition. It has also given us a deeper understanding of the problem of defining a standard library. Hopefully this paper passes some of this understanding along to others.

An implementation of the Standard Library for PC's

The Standard Library has been implemented for the Logitech Modula-2 compiler under the MS-DOS operating system. Some important data structures and procedures as used in this implementation, are described in this article. The library has been implemented for didactical purposes : Modula-2 can be learned using a standardised (let's hope so) and well designed library; additional features for error analysis have been implemented.

The module Files and Binary

The modules File and Binary have been implemented using the Logitech FileSystem module. The type File is defined as a record in the module FileSystem. We give a partial definition :

```
TYPE Flag = (er,ef,rd,wr,ag,txt);
     FlagSet = SET OF Flag;
     BufAdd = POINTER TO ARRAY [0..0FFFEH] OF CHAR;
     Response = (done,notdone,notsupported,callerror,unknownmedium,
                unknownfile,paramerror,toomanyfiles,eom,userdeverror);

     File = RECORD
       bufa : BufAdd;             (* address databuffer *)
       buflength : CARDINAL;      (* length of the buffer *)
       validlength : CARDINAL;    (* number of characters in the buffer *)
       bufInd : CARDINAL;         (* current position in the buffer *)
       flags : FlagSet;           (* state of the file *)
       eof : BOOLEAN;             (* TRUE at end of file *)
       res : Response;            (* result of last operation performed *)
       lastRead : CARDINAL;       (* last word or byte read *)
       ...
     END;
```

Most of the library file operations can be implemented using this data structure. However for the Files.GetFileName operation there are no references to the name of the file that is associated with this data structure. We define a new type as follows :

```
RECORD
  f : FileSystem.File;
  n : ARRAY [0..13] OF CHAR
END
```

The name field "n" will contain the name of the file using the conventional MS-DOS syntax for filenames. This type is defined as an opaque type in the library. An opaque type means that the other modules cannot directly access the fields of the record. A module FileBase for the intermodule communication is defined as :

```
DEFINITION MODULE FileBase;
IMPORT FileSystem;
EXPORT QUALIFIED FileStructure,GetFileVar,PFile;

TYPE FileStructure = RECORD
       f : FileSystem.File;
       n : ARRAY [0..13] OF CHAR
     END;
     PFile = POINTER TO FileSystem.File;
     PFileStructure = POINTER TO FileStructure;

PROCEDURE GetFileVar(f : PFileStructure;VAR p : PFile);
END FileBase.
```

The procedure GetFileVar produces the address of the field FileStructure.f :

```
PROCEDURE GetFileVar(f : PFileStructure;VAR p : PFile);

BEGIN
f := ADR(file^.f)
END GetFileVar;
```

The type File is defined as an opaque type in the Files definition module. In the implementation, this type has been defined as :

```
File = POINTER TO FileStructure;
```

The current implementation of Files allows direct access tot the fields "f" and "n" of this structure. However, the other modules, such as Binary, FilePositions and Text, must access these fields by calling the GetFileVar procedure. This is illustrated by the folowing example :

```
FROM SYSTEM IMPORT ADDRESS;
FROM Files IMPORT File;
FROM FileBase IMPORT PFile,GetFileVar;
IMPORT FileSystem;

VAR file : File;
  f    : PFile;

...
GetFileVar(ADDRESS(file),f);
IF f^.res # FileSystem.done THEN ...
```

Even though referring to a similar data structure, the opaque type File and the type FileBase.PFileStructure are not assignment compatible. Use of the ADDRESS type transfer function can bypass the compiler type checking.

Error analysis

For almost every operation on files the filestate will be evaluated. Detection of an illegal state causes interruption of the program as the standard procedure HALT is called. This procedure generates a memory dump file which can be inspected by a symbolic debugger. Most of the source files are not available in a development environment. To avoid difficulties and for didactical purposes the module Errors has been provided by the implementation.

```
DEFINITION MODULE Errors;
FROM Files IMPORT File;
EXPORT QUALIFIED IOError,Module;

TYPE Module = (files,binary,text,numberio,filepositions,directory,
               standardio);

PROCEDURE IOError(f : File;module : Module);
END Errors.
```

The procedure IOError is called just before HALT is. This procedure prints an error message including a description of the filestate, the module name and the filename, e.g.

"operation on an unopened file in Module Text using abc.xyz"

Use of this imformation allows investigation of the error in the client modules of the library.

Module Text and StandardIO

The module Text includes the operations for manipulation of text files. Redirection of standard input and output will be offered by the implementation of this module. The operations managing this redirection are defined in the module StandardIO.

Redirection of standard input and output

The module Text redirects the standard input and output, updates a logfile and manipulates the error input and output files. The state and the use of these files is set by the operations of StandardIO. Only the definition modules StandardIO and Text define the possible operations. Any direct communication between these modules concerning the data structures of the files is impossible. Therefore, we define an extra module StndBase which performs this communication :

Text StandardIO

StndBase

The module StndBase defines the error, log and redirection files and the state of the use :

```
DEFINITION MODULE StndBase;
FROM Files IMPORT File;

EXPORT QUALIFIED standardIn,standardOut,echoOn,logOn,in,out,log,
                 errorIn,errorOut;

VAR standardIn,standardOut,echoOn,logOn : BOOLEAN;
    in,out,log,errorIn,errorOut : File;

END StndBase.
```

The StandardIO operations consult or change the objects defined with StndBase. The Text operations consult the state of the use and read or write the error, log and redirection files.

The following constraints hold for the current implementation :
- the input of the error file is always read from the terminal;
- the output of the error file is always written to the terminal.
In both cases no real file is involved.

The state "eol"

The function EOL returns TRUE if the last operation was not performed due to the occurrence of either an end of line or an error. The state of a file "alfa" in maintained in the field alfa.flags. An error state can be tested by

```
IF FileSystem.er IN alfa.flags THEN ... END
```

Since the eol state is not provided in flags, we define a new type Flag :

```
TYPE Flag = (er,ef,rd,wr,ag,txt,eol);
```

and the type ExtFlagSet

```
TYPE ExtFlagSet = SET OF Flag;
```

For the implementation of the type ExtFlagSet we use the property that some of the bits of the internal representation of FileSysten.FlagSet are not used. The procedures SetEol and ResetEol manipulate the "eol" state of a file. The type identifiers FileSystem.FlagSet and ExtFlagSet are also used for type transfer.

```
PROCEDURE SetEol(file : File);
VAR g : ExtFlagSet;
    f : PFile;

BEGIN
GetFileVar(ADDRESS(file),f);
g := ExtFlagSet(f^.flags);
INCL(g,eol);
f^.flags := FileSystem.FlagSet(g)
END SetEol;


PROCEDURE ResetEol(file : File);
VAR g : ExtFlagSet;
    f : PFile;

BEGIN
GetFileVar(ADDRESS(file),f);
g := ExtFlagSet(f^.flags);
EXCL(g,eol);
f^.flags := FileSystem.FlagSet(g)
END ResetEol;
```

The eol state of a file can be evaluated using the statements

```
GetFileVar(ADDRESS(alfa),f);
RETURN eol IN ExtFlagSet(f^.flags)
```

Module NumberIO

This module has been implemented using the procedures Text.ReadString and Text.WriteString and the conversion procedures of module Convert. The base parameter of the ReadNum and WriteNum procedures is restricted to the interval [2..16].

Module Terminal

The Terminal module is based on the Logitech Terminal module. To avoid conflicts between these modules, the library module has been renamed to TermnlIO. The procedures for screen manipulation are implemented for an IBM or compatible PC using the ANSI.SYS driver program.

Module SimpleIO

Module SimpleIO

The module SimpleIO has been implemented using the procedures of the module Text and the Logitech module InOut. We used the InOut implementation instead of the TermnlIO module so that redirection with the command line can be specified with the operators "<", ">" and ">>". For example the execution of the command

```
m2 prog <in >out
```

when using the operations of SimpleIO, has the same effect as the explicit programmed redirection, which is realised with the statements :

```
Open(input,"in",textMode,readOnly,state);
SetInput(input);
Open(output,"out",textMode,readWrite,state);
SetOutput(output);

...
Close(input,state);
Close(output,state);
```

Module Program

This module has been implemented using the Logitech modules Program and System. To avoid conflicts between the Logitech Program module and the library module, this last one is renamed ProgramCall.

Module String

String is based on the Logitech Strings module. Additional tests have been added to evaluate the success parameter of the library procedures.

Some conclusions

The library was implemented on a PC within a few weeks and works very well. The linking speed of programs is rather slow due to the lot of standard and Logitech modules involved. The program listing of the implementation takes about 50 pages.

E. Verhulst
Software Engineer
Langbaanvelden 140
B 2100 Deurne
BELGIUM

EDITORIAL

The MODUS Quarterly #4, November 1985

The MODUS meeting

We met in Menlo Park. Standards work, library suggestions, compilers and applications were all subjects of discussion. Several papers from the conference are reproduced here. "NewStudio: Engineering a Modula-2 Application for Macintosh" is scheduled to appear in the February issue. If you need it sooner, write to me for a copy now.

Timeliness

I have revised the schedule of deadlines and publication. The details are on the inside front cover. This might be termed defeat.

Prizes for articles/suggestions

Notwithstanding his flaws, your editor now has begun to charge MODUS for the task of putting out each issue. This revenue permits offering two prizes per issue. The winners of the "Best Article" and the "Best Suggestion" prizes will each receive a full year's membership in MODUS as well as a tacky "Certificate of Honor" to express the gratitude of your editor (and perhaps other MODUS members) for their contributions. Send your vote and your suggestion to the editor. Do it today!

Mailing problems (USA)

We found another way to get sub-optimal mailing for issue #3. This time the "special first class" rate was used which apparently means that forwarding instructions are not honored. Some of these copies were returned, but we suspect others were not. If you are missing issue #3, please contact your administrator: see inside front cover.

Last chance to RENEW

If you are one of the many readers who have not renewed your membership in MODUS, then YOU WILL NOT GET THE NEXT ISSUE. You are forewarned.

rhk [Just try to get a square inch past me.]

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

Modula – 2 Compilation and Beyond

D.G.Foster[1)]

Geneva, 29 August 1985

Presented at Modula – 2 Technical Conference, Menlo Park, CA. 5 – 6th Sept. 1985.

[1)] CERN, DD Division, Geneva, Switzerland

1. Abstract

The implementation of a new Modula – 2[1] (3). cross compiler is described. The compiler creates, and maintains, a simple "module base" file that contains information about separately compiled modules. The information within the module base file can be used by a small number of portable tools to aid program development and maintenance. Tools that provide automatic recompilation of modules, graphical representation of multi – module systems and prelinking functions are described.

In addition the implementation details of a multi – language cross compiler suite are discussed with reference to the specific requirements of Modula – 2.

2. Introduction

Programming "stand alone" microprocessor systems poses some problems for software development. One is obliged to prepare programs on a host machine and load the resultant object modules into a target microprocessor to be executed. The minimum software required is a cross assembler, although the use of cross compilers for high level languages is becoming more widespread. In practice the cross compilers do not replace assembler programming especially where direct access to hardware dependent features are required. Therefore an important requirement of a cross software system must be the ability to mix, with a given high level language, either assembler routines or routines produced by other cross compilers.

We have produced a cross software system to support the Motorola 680x0 series of microprocessors. The cross software has been described in (1). The rest of this paper will concentrate on one aspect of the cross software, namely the implementation of the Modula – 2 compiler (2).

[1] Modula – 2 was designed at the ETH, Zurich.

3. Compiler requirements

The following were identified as the principle requirements of the Modula−2 compiler:

1. It should be an integral part of the cross software system. This ruled out available compilers and fixed the implementation language as Pascal.

2. It should support the full Modula−2 language, not just a subset. This forced a multi−pass compiler configuration.

3. It should obey the calling convention requirements to provide inter−language mixing, and should provide symbolic debug tables (1). This is satisfied by sharing a common code generator between the cross compilers.

Figure 1 shows the chain of processes that take place to produce a load image from a source program. It should be noted that the identity of the originating language is lost after code generation. In our system the link editor is a general tool with no language specific features. I will come back to this point later.

Having provided a new "front end" to the common code generator, some additional requirements became clear:

1. To find a technique to order module bodies correctly and have them initialised before the main program.

2. To provide the essential tools for a programmer to create and maintain a collection of related modules.

4. Compiler design

I will not attempt to provide complete justification for each stage of the design, instead I will concentrate on describing the final product which I will call "Mod68k". To avoid confusion the "front end" can be considered as consisting of pass0 and pass1 that feed the code generator (pass2).

It was decided to produce a recursive descent implementation of Modula−2 that would be a two pass front end producing the intermediate format of the common code generator. Multi pass compilers tend to have a lot of almost duplicated code in each pass, with additional code to write out and read in the interpass formats. In order to avoid this, Mod68k uses "in memory" symbol table information that is created as an unbalanced binary tree during pass0 and subsequently used during pass1. Thus, pass1 can be considered as performing the actual semantic analysis after pass0 has passed declaration information to it.

The functions of pass0 and pass1 can be coalesced, almost completely, such that very little special code is required to differentiate between the two passes. This is at the expense of some speed since it implies that the source code is read, and parsed, twice. Simply then, the compiler is organised as a single pass recursive descent compiler. Complete compilation is attempted by calling the main program twice with just a boolean variable to distinguish between pass0 and pass1. The compiler can also be instructed to attempt complete compilation in 1 pass only; in this case forward references would be flagged as errors.

In multipass mode the symbol table information for each scope is organised as a linked list and as outer scopes are re−visited and more declarations found, these are simply added to the tree of that scope level. Complete compilation is then attempted by pass1 using the, now complete, symbol table information at each scope level.

This recursive descent compiler will compile a module. A compilation unit is defined as being an implementation, definition or program module. However in terms of compiler organisation the local module may be considered as being part of that list. To compile a local module the current state of the compilation of the parent implementation, program or local module is saved and the main program of the compiler called recursively. After compilation of the local module the parent state of compilation is restored and the scope of variables exported from the local module extended.

So far we have not said anything about importation of modules. From the above ideas it is a small extension to allow the importation of modules. The symbol table information needs to be made available for the importing module and this can be done in a variety of ways. One technique is to produce a symbol table file after compilation of a definition module which contains the, already processed, declarative information expressed by the definition module. To avoid the extra files required to save symbol table information, Mod68k will simply recompile all definition modules from which objects are imported. This is achieved in an identical way to the compilation of local modules. That is, the current compilation state is saved, the appropriate file opened and the main procedure of the compiler called recursively. All declarative information required during the subsequent compilation is then available from the preserved symbol table trees.

Visibility of objects and scope rules are obeyed by copying objects between appropriate symbol table trees.

5. The module base concept

The previous chapter introduced the idea of recompiling definition modules when objects from them were required. The source of these definition modules need to be made available to the compiler. This is done by using a so called "module base".

Initiating the compiler causes it to read information from one file, the module base. This file may be empty, or it may contain information relating to previously compiled modules. For each module compiled an entry in the module base is either created (the first time) or updated. Figure 2 indicates the contents of this module base.

Each entry has an entity number that is used for reference purposes within the module base, and the module name of the entry. Attached to each entry are the file names, dependency lists (of entity numbers) and checksums for the definition and implementation parts of the module. The compiler finding an IMPORT statement can therefore find the physical file of the definition module and process it.

There is an assumption here, that the host Pascal system used to implement the Modula−2 compiler be capable of opening a file whose name is contained within a variable. The file name in the module base is usually a path name of some description. This will vary from system to system, but to enable the module base files to be copied within a given file base, the file name should uniquely identify the file within the file base. Since one normally passes to the compiler an incomplete path name, it is the command script invoking the compiler which expands the name to the full path. The compiler itself merely passes the name from the command line to the module base.

5.1 Module invalidation

Recompiling a definition module requires that all modules which depend on it be recompiled also. The compiler, having compiled a definition module, determines from the dependency lists all dependent modules. These modules are marked in the module base as being invalid. In practice this means the date/time field for an invalid entry is set to null.

6. Tools

The information in the module base can now be used, by a number of tools, to provide the facilities required by a programmer to maintain his multi module system easily These tools consist of a Pascal program which reads the module base and performs the necessary processing. The result is passed to a command script which performs the necessary operating system dependent functions. In this way the "non portable" code has been kept to a few lines of command script.

In the following discussions I shall use the terminology (a) < − (b) to mean that module "a" imports from, and therefore depends on, module "b".

6.1 Recompilation

The recompilation tool (Modrecom) enables the programmer to automatically recompile invalidated modules. Modrecom may be invoked with a filename argument, in which case Mod68k is invoked to compile the given file, or with no filename argument. In either case Modrecom checks the module base for invalid modules and determines the order of recompilation. For definition modules this order is important. Consider the system definition (a) < – definition (b) < – definition(c) in which recompiling definition module "c" has invalidated both "a" and "b". The optimal order of recompilation is "b" followed by "a". If "a" is recompiled first then recompiling "b" would again invalidate "a" requiring it to be compiled a second time.

Additionally, modrecom may be instructed to compile all definition or implementation modules that appear in the module base. This is very useful when installing new compiler versions.

6.2 Graphical representation

The reason for producing this tool (Modgraph) is somewhat historical in that it produces the same representation as was created by hand before this tool became available. This is simply a grid representation where "I" and/or "D" at the junction of a horizontal and vertical module indicate that the horizontal module imports from the vertical definition module in either the implementation or definition module. Given the information in the module base other, more aesthetic, representations could be generated, but we have not found the necessity for this.

6.3 Link editing

It is now appropriate to discuss the functions which must be performed during the link edit phase. This is included in the tools section because it is the tool "Modlink" that orders the modules for the link editor. This order is important since this determines the order in which module bodies will be initialised in our system. Modlink determines, from the dependency lists, which modules should be initialised first. Given the possible system: implementation (a) < – definition (b), the implementation module of "b" must be initialised before that of "a". Variables assigned initial values in the implementation module of "b" may be used in the module body of "a". However in conjunction with the above system, the following is possible: implementation (b) < – definition (a). Now the order of initialisation of module bodies is undefined. In this case Modlink will issue a warning, and will produce, if requested, a complete list of all such cyclic dependencies.

6.4 Concatenation

The module base is recommended to be used to reference an entire program, containing just one program module, or as a "library" containing references to a number of related modules. As a starting point several "library" module bases may be concatenated to enable access to a variety of facilities. For example one might consider concatenating the module bases for I/O, file handling and transcendental functions. Thereafter, any program written may reference these facilities without any need to know their positions in the file base.

The tool "Modcat" has been provided to concatenate module bases and perform the necessary renumbering of entity numbers and dependency lists. In a given module base, each entry must have a unique entity number.

7. Consistency checking

There are two points at which consistency checking should be performed in the software chain shown in figure 1 . These are distinct because the acts of creating separate object modules and of linking them to create a complete program are disjoint. The compiler may check as it is using a definition module that the interface has not changed since it was compiled by verifying that the calculated checksum is identical to the corresponding checksum in the module base. Within one module base any change of an interface module will flag dependent modules as being invalid. However consider the following sequence of events: A library is created with a module base. This module base is copied as a starting point for a software project. A program is generated correctly. If now the library is changed BEFORE the new program is link edited, but AFTER it has been compiled then this will not be detected. The reason for this is that invalidation of dependent modules only takes place within one module base. An extra verification is therefore required in this case before link editing. One approach is to have an extra tool that simply performs the same checksum verification as the compiler. This may be run prior to link editing as the object modules are being ordered by modlink. Essentially it would check that the library files referenced by the current module base have not changed since the library module base was copied.

8. Module body initialisation

Each object module produced after the compilation of an implementation or program module may be considered as consisting of 3 sections.

1. The CODE section contains the executable statements and long constants.

2. The DATA section contains the global variables.

3. The BODY section contains one word holding the address of the entry point of the module body.

All object modules that constitute a complete program are initially ordered and subsequently concatenated by Modlink and passed to the link editor. The link editor joins like sections together as it reads the concatenated file and performs the usual functions of linkage edition.

The important point to note is that the BODY section is allocated before the CODE section and the first module allocated in the CODE section is the run time system. A memory map of a Modula−2 program will look like Figure 3 . Adjoining the run time system, in memory, is now a "stack" of entry points to all the module bodies. It is a simple matter for the runtime system to execute each of the module bodies, as a procedure in reverse order of loading. The last module body to be executed in this way is the main program (Modlink arranges this).

This approach to module body initialisation does not require a special link editor, dedicated to Modula−2, to be ported with the compiler. It does require the tools to be ported, but these each consist of a few hundred lines of "vanilla" Pascal.

9. Some implementation information

The compiler consists of less than 9000 lines Pascal, for the front end, and less than 6000 for the code generator. The intermediate format is a low level linearised tree structure that is recreated into a data structure by the code generator. For systems with sufficient memory this tree could be kept in place with the pointer to its root passed from the front end to the code generator. Table 1 indicates the sizes allotted to the basic types by the compiler.

The default REAL is the larger precision REAL, in this case 6 bytes, with REAL32 the identifier for the smaller precision. For consistent naming conventions these should be LONGREAL and REAL respectively.

Table 1: Basic sizes allotted by the compiler

TYPE	*SIZE in BYTES*
INTEGER	2
CARDINAL	2
LONGINT	4
LONGCARD	4
BITSET	2
WORD	2
ADDRESS	4
REAL	6
REAL32	4
SET OF	Up to 32 bytes

At present, function procedures may return structured types – This is the only way to make exported structures read only.

The compiler has been ported to VAX computers running UNIX and VMS, as well as NORD 500 computers running SINTRAN. Workstation ports include APOLLO, SUN and CADMUS. A port to an IBM system running VM is in progress.

10. Conclusion

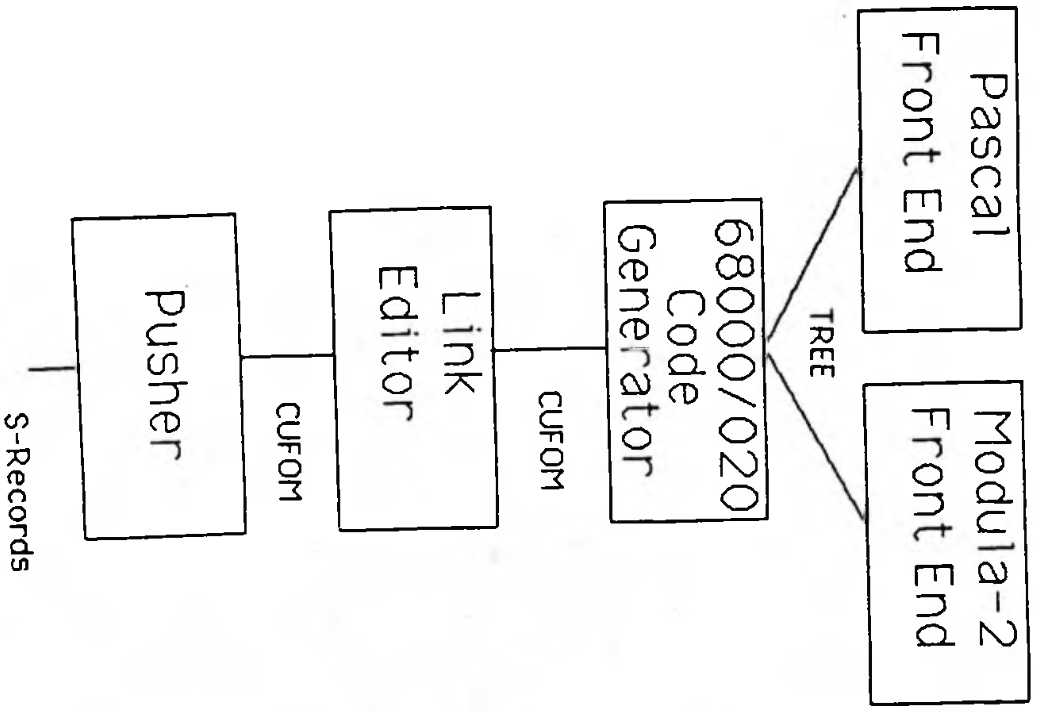
The module base concept provides a portable way of accumulating information on separately compiled modules. This removes the need to create special symbol files and ensures that the interface (i.e. definition modules) seen by both the programmer and compiler are identical. The information in the module base can then be used by portable tools to build a programming environment for Modula – 2. This environment enables a programmer to maintain his separately compiled modules easily and goes beyond simply providing a compiler. Hence the title of this talk.

References

(1). Developing Programs for the Motorola 68000 Microprocessor at CERN, J. Blake, H. von Eicken and D. Foster. Europhysics Conference on "Software Engineering Methods and Tools in Computational Physics". August 1984.

(2). Separate Compilation in a Modula−2 Compiler, D. Foster. Software Practice and Experience. − to be published.

(3). Programming in Modula−2, N. Wirth. Springer−Verlag publication, 3rd Corrected Edition. 1985.



Producing a Load Image From a Source Program

Figure 1

PAGE 45

Module base

- *Entity number*
- Module name
- File name of the DEFINITION module
- DEFINITION module compilation Date/Time
- Checksum
- Dependency list of the DEFINITION module
- File name of the IMPLEMENTATION module
- IMPLEMENTATION module compilation Date/Time
- Checksum
- Dependency list of the IMPLEMENTATION module

Figure 2

Memory Map of a Multi-Module System

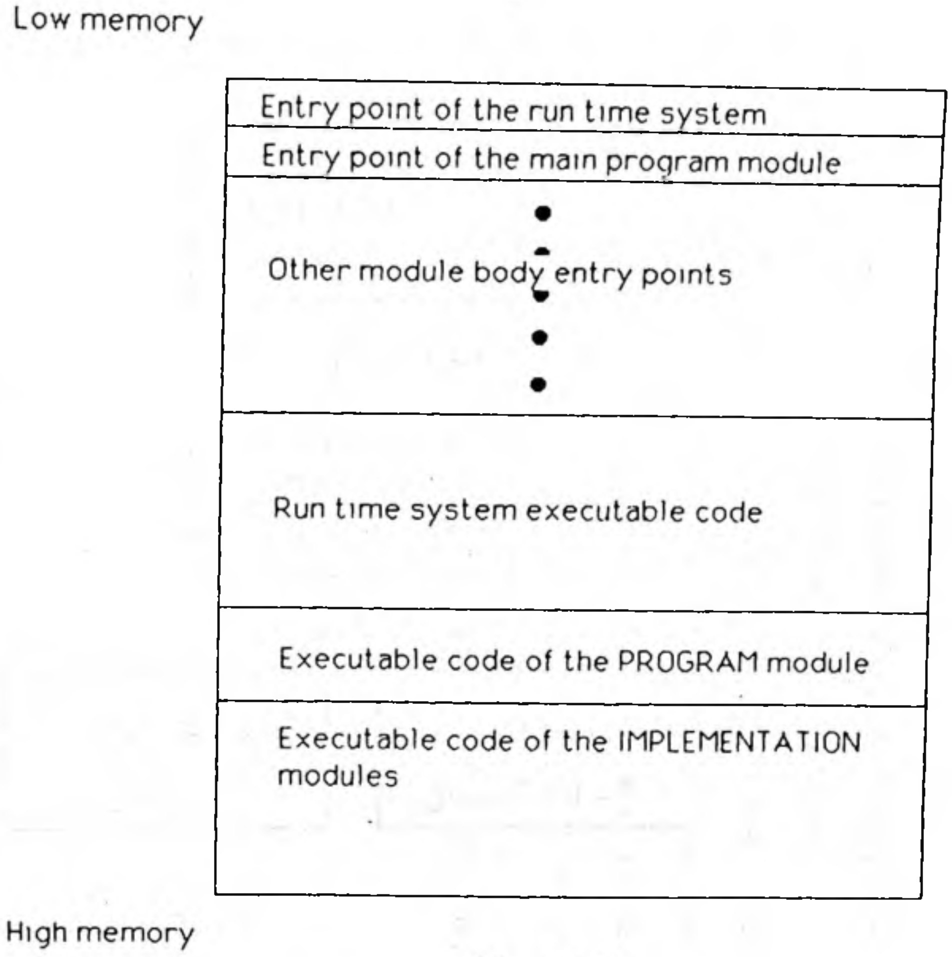

Low memory
Entry point of the run time system
Entry point of the main program module
Other module body entry points
Run time system executable code
Executable code of the PROGRAM module
Executable code of the IMPLEMENTATION modules
High memory

Figure 3

Modula-2 Processes - Problems and Suggestions

Roger Henry

Computer Science Group
University of Nottingham
Nottingham NG7 2RD
UK

written version of paper presented at the
Technical Meeting of the Modula-2 Users' Association
Menlo Park, California
September 5th - 6th 1985

Introduction

Modula-2 is I assume our (currently) favourite language. While its very name forces us to recognize the influence of its immediate predecessor, *Modula*, perhaps the comparisons made most frequently are those with Wirth's best known language *Pascal*. This is understandable since *Modula-2* and *Pascal* both have potential as general-purpose programming languages. *Modula* was aimed solely at the very specific applications area of programming embedded industrial and scientific real time computer systems (and that primarily for the DEC PDP-11 range of machines). *Modula-1*, as perhaps we should now call it, still has its loyal users but to the best of my knowledge it has not had the benefit of a users' association like Modus, and neither has it had its name included in the title of a journal to which present company may subscribe! One definite attraction for me to the second bearer of the name *Modula* is that my involvement with it has given me this opportunity to visit California.

Now I admit to using Modula-1 before Pascal, but then I also admit to reading *The Robots of Dawn* before either *The Caves of Steel* or *The Naked Sun* and there are definite advantages. A programmer's initial view of a language seems to be dominated by the languages he or she has studied first. Thus on discovering Modula-2, a Pascal programmer may take most notice of the addition of modules and the subtraction of high-level input/output. When I obtained Modula-2 from Zurich in 1980 I worried about the subtle changes in the module construct but took most notice of the subtraction of the **process** keyword together with the associated operators and types.

I was sensitized to the apparent removal of built-in processes because I had been championing the use of *Modula* for the on-line control of experiments in what was then my home ground: the Psychology Laboratory. Typical applications included the study of visual performance using computer generated dynamic displays for output, and key presses or eye-movement monitoring for input. This was work which had previously relied on tricky assembly language programming. Even then it had been found that the use of multiple execution threads often simplified the solution to a control problem. Sometimes a demand-driven scheduling approach enabled experiments which would otherwise have been infeasible within the real-time constraints. Instead of a single process needing to juggle to try and keep more than one activity going at once, the scheduler could route the flow of control to suit the dynamic needs of each task. Separation into processes also facilitated the modularization of large programs.

Multiprogramming in Modula-1

It is worth pausing to recall the multiprogramming facilities of Modula-1 since they have served as a model for a proposed standard module in Modula-2. The relevant constructs are process declarations, process statements, interface modules and signals. For the PDP11 implementation at least, there are also device modules, device processes and the *doio* statement. These related sets of facilities are illustrated by Examples 1 and 2.

```
module Example1; (* semaphores in Modula-1 *)

 interface module resourcereservation;
  define
   semaphore, P, V, init;
  type
   semaphore =
    record
     taken: Boolean;
     free: signal;
    end;

  procedure P(var s: semaphore);
   begin
    if s.taken then wait(s.free) end;
    s.taken := true
   end P;

  procedure V(var s: semaphore);
   begin
    s.taken := false;
    send(s.free)
   end V;

  procedure init(var s: semaphore);
   begin
    s.taken := false
   end init;

 end resourcereservation;

 var
  turn: semaphore;

 process doit(x: integer);
  begin
   while x > 0 do
    P(turn); (* now use resource *) V(turn);
    x := x-1
   end
  end doit;

 begin
  init(turn); doit(20); doit(15)
 end Example1
```

The program of a process is distinguished from a procedure by the syntax of the declaration but not by the form of the call. Notice that parameters may be passed when processes are started. The activating process statements are restricted to the body of the main program. This simplifies the allocation of workspace. When control reaches the end of a process, the process goes out of existence - but its workspace is not reclaimed.

The interface module is intended as a construct to maintain mutual exclusion between processes in access to shared variables. It therefore corresponds to the *monitor* of *Concurrent Pascal* The assurance is that there be no interleaving of statements belonging to the procedures of the same interface module except during the execution of the synchronizing operations on signals. So once a process executing **P** in Example1 has found **s.taken** to be true, another process cannot intervene before the inevitable wait for **s.free** - at least not in a way which will allow a call of **V** to invalidate the condition. This assurance may be given as a result of the method of process scheduling.

In Modula-1, *signal* is a kind of sub-standard type. It can be used in further type declarations and in variable declarations, although the resulting objects are not variables in the sense of having values which may be assigned. Operations are limited to the standard procedures *wait* and *send* *Wait* delays the calling process until there is a corresponding call of *send* *Send* transfers control to the process which has been waiting longest for the given signal or simply does nothing if no processes are waiting for it. When a process waits, another ready process is selected which then returns from its call of *send* The standard predicate *awaited* may be used to test if any process is waiting for a given signal.

In practice, the only difference between interface modules and ordinary modules is that interface procedures must not call imported procedures. This excludes the possibility of interleaving being induced by external calls of the signal operations. The whole strategy is very conservative in the sense that mutual exclusion occurs when it is strictly not necessary - for example between calls of **P** and **V** on different semaphores. The security of semaphores is guaranteed by the intransparency of exported types in Modula-1. Nowhere outside the module can the program directly alter the *taken* field for example, since the structural details are not available to the programmer.

Device processes are processes declared and initiated entirely within device modules. Essentially they serve as interrupt handlers. The heading of a device process shows the vector address of the interrupt they are designed to handle and the heading of the device module shows the priority level to which the processor is to be raised. This priority should correspond to the interrupt priority of the device so that interrupts are fended off until the device process has issued a synchronizing 'wait for interrupt' request. This it does by executing the standard procedure *doio* Only one instance of a device process may be started because of the way in which it is associated with a fixed interrupt vector. All of this is illustrated by the module **realtime** in Example 2.

The interface between a regular process and a device process is provided by exported procedures of the device module. Mutual exclusion is assured by raising the processor priority to the priority of the device module during the execution of device procedures. An interrupting device process is prevented from upsetting mutual exclusion in interface modules by enforcement of the rule that no external procedure may be called from within a device module.

Signals may be used for synchronization between a regular process and a device process (but not between device processes). Because of the implied higher priority , the semantics are modified so that sending a signal from a device process simply marks the receiving regular process as ready to run. The device process continues until it decides to wait, either for a signal from a regular process or for an interrupt from a device. Control then returns to the process which was interrupted.

```
(* Example 2 - device modules and processes *)

device module realtime[6];
 define time, tick, pause;
 var
  time: integer;
  tick: signal;
  LCS[177564B]: bits; (* PDP-11 line clock status register *)

 procedure pause(n: integer);
  begin
   while n > 0 do
    wait(tick); dec(n)
   end
  end pause;

 process clock[100B];
  begin
   LCS[6] := true;
   loop
    doio; inc(time);
    while awaited(tick) do send(tick) end
   end
  end clock;

begin
 time := 0; clock
end realtime
```

Multiprogramming in Modula-2

As a systems implementation language, Modula-2 has been designed to allow the actual construction of schedulers such as the one which underlies Modula-1. If we take a layered approach to system building, the bottom layer is either the actual hardware, or an operating system, as 'fixed-up' by the run-time system to make the underlying machine suitable for executing Modula-2 programs. Higher layers are all expressed as Modula-2 programs, that is using the same notation, but with further operations available through the published definition modules of lower layers. Some layers may provide facilities which are intended to replace those of lower layers as far as higher layers are concerned. A process scheduler will occupy one of these layers.

An obvious first step in evaluating Modula-2 for real-time control applications is to implement a Modula-1 style scheduler. In fact Wirth has done this for us in his proposed standard module *Processes*. Comparisons can therefore be made quite easily. However, for me, and I hope for you, Modula-2 offers the prospect of experimenting with other approaches. It may turn out that a different design would have advantages as a middle layer in a system built with Modula-2. We

PAGE 50

should bear in mind that an important motivation for having a standard module is to promote the portability of the code in the client modules of higher layers. The implementation of a layer may be system dependent but, as a general rule, we should aim to achieve portability at as lower a layer as possible.

Low-level facilities

Let us proceed then by looking at the tools offered to us for building a process scheduler in Modula-2. We rely, of course, on the syntactical facilities of the language - the ability to declare self-initializing modules, procedures, types, opaque types, variables and so on. What else we can rely on depends upon the status of the pseudo-module SYSTEM. In the latest version of the Modula-2 report, Wirth states that the facilities exported from the module SYSTEM are specified by individual implementations. "Normally, the types WORD, ADDRESS, and the procedures ADR, TSIZE, NEWPROCESS, TRANSFER are among them." It is the latter two which interest us in the present context since they provide the essential coroutine mechanism for creating separate threads of control and transferring execution between them. Originally, coroutines were referred to by values of the type PROCESS which was also exported from SYSTEM. Now, as if to press home the low-level nature of coroutines, PROCESS has been replaced by ADDRESS.

```
PROCEDURE NEWPROCESS(
                P: PROC;
                A: ADDRESS;
                n: CARDINAL;
                VAR p: ADDRESS);
```

prepares a new coroutine with program P and workspace starting at A of length n. The returned ADDRESS value, p, refers to the coroutine in its initial state. For correct programming, it is vital to realize that the same coroutine may subsequently be referred to by a different ADDRESS value as its state changes. These values really only have meaning while the coroutine is not executing. The latest value is stored for us when one coroutine explicitly transfers control to another by calling

```
PROCEDURE TRANSFER(VAR p1, p2: ADDRESS);
```

The value of p1 is the one to use later as p2 if control is to be returned to the calling coroutine. This applies equally to the main program as to any dynamically created coroutine. The possibility of building a scheduler on top of these low-level facilities rests on the meaningfulness of these *values* and of their assignment to designated variables.

There is some debate as to whether these routines belong in SYSTEM. The general intention is that this module be the repository of facilities close to the computer being used. Unless you are using a Lilith, this does not seem to encompass NEWPROCESS and TRANSFER. The run-time system provides these mechanisms for the benefit of Modula-2 programs. It also handles the termination of the program should the procedure nominated in NEWPROCESS attempt to return and sets up the main program to look like a coroutine itself. All of this should be possible to implement on any machine capable of supporting the rest of Modula-2. If PROCESS were to be kept, and made a standard type, only the way in which the workspace is currently specified would prevent NEWPROCESS from being raised to the status of a standard procedure. There would be no such problem with bringing TRANSFER 'out into the open'. A precedent for such migration has already been set for the procedure SIZE.

I would argue against moving the access of coroutines out of SYSTEM, provided that it was clearly understood that the mechanisms decribed in the report are to form a model to be followed by implementors as closely as possible. This would allow the low layers of a system written in Modula-2 to be portable from one implementation to another. I believe that NEWPROCESS and TRANSFER are examples of facilities that should be replaced at intermediate layers and never used by the higher layers. It is therefore not appropriate to make them into standard procedures which are available everywhere without being explicitly imported.

The standard module *Processes*

What should come at the intermediate layers? Following Wirth's book and the original 'yellow book' report from ETH, many implementations provide the Processes module for synchronization between regular processes. This is modelled on the regular process facilities of Modula-1 as can be seen from the published definition module

```
DEFINITION MODULE Processes;
 TYPE Signal;
  PROCEDURE StartProcess(P: PROC; n: CARDINAL);
  PROCEDURE SEND(VAR s: SIGNAL);
  PROCEDURE WAIT(VAR s: SIGNAL);
  PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
  PROCEDURE Init(VAR s: SIGNAL);
END Processes.
```

The book shows how this is implemented by creating a descriptor for each process and linking it into a ring. Descriptors of processes waiting for a signal are linked into a queue for that signal The initialization part sets up the ring to include only a descriptor for the main process.

Some differences are forced upon us by the language changes. Thus procedures and processes are now distinguished only by the form of the call: **P or StartProcess(P, n)**. I suspect that it could be argued that this is actually an improvement as far as program readability is concerned. The removal of parameters to processes, on the other hand, represents a significant loss of convenience. The procedure **P** has to be written to copy global variables into local variables which makes the code less easy to understand. The relaxation of the restriction that processes cannot be started in procedures or processes does help in structuring programs.

In Modula-1 we were used to the fact that a process would simply go out of existence if it reached the end of its program. In the case of the usual implementation of the module Processes in Modula-2, if the procedure of a process attempts to return then the entire multiprogram goes out of existence. This is simply because the procedure **P** given to **StartProcess** is used directly in a call of NEWPROCESS. A partial fix would be to have **StartProcess** employ a local procedure of its own as the program of all processes and then by copying into and out of a global variable have it call **P**. On return the process could unlink its descriptor from the ring and arrange for the next ready process to be selected. The workspace could also be reclaimed but not by the terminating process! The fix is partial because there is no getting away from the fact that the end will come when the main process comes to the end of its program. With the standard Processes module, the only way for a process to give up and leave the work to its coprocesses is to introduce the overhead of waiting for a signal which never comes. It perhaps should be called **tomorrow**.

Most of the other changes have detrimental effects on program security. Thus the user must specify the workspace size and risk disaster if too small a value is chosen. Unfortunately, the safe value for this parameter may be the only non-portable aspect of a program which uses Processes. This could be ameliorated to some extent if the semantics were changed to make the given size an increment over and above some minimum chosen by the implementation of Processes. Scaling the size by SIZE(CARDINAL) might also help but is clearly not a completely satisfactory solution to this problem.

The user must now remember to call **Init** once only for each declared signal. As with all hidden types, signal values can be tested for equality and assigned even though this is of doubtful meaning. For example, are these equivalent code fragments?

```
IF SomeTest() THEN WAIT(s1) ELSE WAIT(s2) END;

IF SomeTest() THEN s := s1 ELSE s := s2 END;
WAIT(s);
```

The answer depends upon the implementation of signals. With Wirth's code, signals point to the head of a list of descriptors of waiting processes and so the second statement sequence would leave s1 or s2 with an incorrect value. An equally valid implementation would make signals pointers to a pointer to the head of a list of descriptors which are therefore never changed by **WAIT**. A hint against this is given by the choice of VAR parameters for the signal operations, but it would seem safer for the language to have ruled out altogether tests of equality and assignments on hidden types.

The role of the interface module is now taken by an ordinary Modula-2 module. There can therefore be no enforcement of the rule that interface modules should not call non-local procedures. Beware of the fact that the logic of the argument which assures mutual exclusion in interface procedures may be subverted by calls to external routines.

Interrupt handling

Modula-2 is not a contender for real-time control applications without a method of handling interrupts (confusingly called exceptions on some machines). Indeed it is the linking to external events which breathes life into a multiprocess system implemented on a single processor - as is assumed to be the case for Modula-2 (This is not to deny that quasi-concurrent solutions to some non real-time problems can be attractive.)

There is no doubt that the details of interrupt handling are going to be system dependent. There will be some implementations for which there are no interrupts to handle. In other cases the interrupts will be generated by the hardware but may either be specified by number or by a vector address. I would like to see implementations where interrupts were generated by completion of asynchronous transfers handled by an underlying operating system. Wirth has shown us how an interrupt handler can be modelled as a cyclic process synchronizing with an externally generated event. In Modula-1 this synchronization was expressed by the *doio* statement within special *device* processes. In Modula-2 for the PDP-11 handling hardware interrupts there is the lower-level procedure

```
IOTRANSFER(VAR p1, p2: ADDRESS; va: CARDINAL);
```

which stores a reference to the current state of the calling coroutine in p1 and transfers to the coroutine referred to by the value of p2. The run-time system arranges matters so that when an interrupt occurs through the vector va, a reference to the state of the interrupted process is stored in p2 and the IOTRANSFER returns to its calling coroutine. There is a full process switch on receipt of the interrupt and so no real distinction exists between regular and device processes. The only requirement is that IOTRANSFER be called in a module with a specific priority corresponding to the priority assigned by the hardware to the interrupting device. More precisely, the processor priority must be kept at this level while interrupts are enabled and the handling process is executing.

The generality of specifying the vector address at each call of IOTRANSFER may seem to be a small prize to win for the price of setting up the vector dynamically each time. The comparison with traditional interrupt handlers can be misleading as there the interrupt routine uses the stack of whichever process it interrupted. Most of the extra overhead comes in arranging to switch stacks rather than in setting up the vector. This has the rich prize of allowing local variables to remain in existence between interrupts.

It comes as no surprise that IOTRANSFER must be imported from SYSTEM, although it is interesting to see that some implementations for processors other than the PDP11 have been able to keep a compatible interface. There may be problems with some processors in ensuring the indivisibilty of the transfer operation from the interrupted to the interrupting process.

```
MODULE Printer [4];  (* Example 3 *)
  FROM SYSTEM IMPORT
     NEWPROCESS, TRANSFER, IOTRANSFER, LISTEN,  ADDRESS, WORD, ADR;
  EXPORT  Print;
  CONST  N = 32; intEnable =6; intVec = 64B;
  VAR
    n: CARDINAL;  in, out: [1..N];  waiting: BOOLEAN;
    buff: ARRAY [1..N] OF CHAR;
    user, driver: ADDRESS;
    wsp: ARRAY [1..100] OF WORD;
    statusReg [177564B]: BITSET;  outBuf [177566B]: CHAR;

  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
    INCL(statusReg, intEnable);
    LOOP
      IF n = 0 THEN waiting := TRUE; TRANSFER(driver, user) END;
      outBuf := buff[out]; out := out MOD N +1; DEC(n);
      IOTRANSFER(driver, user, intVec);
    END
  END Handler;

BEGIN
  n := 0; in := 1; out := 1;
  NEWPROCESS(Handler, ADR(wsp), SIZE(wsp), driver);
  TRANSFER(user, driver);
END Printer
```

PAGE 54

ted process is
ocess switch
nd device
specific
device
bled

Many commercially available systems provide no higher-level facilities akin to those of Modula-1 device processes. It turns out that the coroutine transfer model is quite useable for device handling. This is because the interface procedure called by the user process knows the identity of the device process with which it interacts. This is not the case for regular interface modules where the anonymity of processes waiting for signals is more appropriate. A common case is illustrated by Example 3.

The use of the Boolean flag **waiting**, allows the user to access a slot in the buffer as soon as the character has been taken out by the driver but avoids a TRANSFER being made while the driver is still waiting for an interrupt. Without this flag, the user could instead test for the case **n = 1** but then the statement **DEC(n)** would have to be postponed until after the interrupt had been received.

It may seem tempting to turn on interrupts in the module body along with the initialization code. After all, the body does execute at priority level 4. Unfortuneately there are implementations, including those for the PDP11, in which NEWPROCESS sets the initial priority to 0. On the first transfer to the process, the processor priority will be lowered to 0 before the code of the procedure gets a chance to raise it back to its proper level. Possible fixes would be to add a parameter to NEWPROCESS to give the initial priority, or to make the initial priority the same as that of the caller.

To guard against the possibility of spurious interrupts occurring when no IOTRANSFER has been issued, it would be safer to enable interrupts before each IOTRANSFER and to disable them again immediately afterwards.

The LISTEN procedure momentarily lowers the processor priority to let in pending interrupts. It should not be possible for you to write your own version of LISTEN in Modula-2 - unless you took the trouble to transfer to a priority level 0 process which then immediately transferred back to you. The language rules imply that a process currently executing a procedure (declared in a module) at priority level n must not call a procedure at priority level m < n. This is to protect the mutual exclusion afforded by a high priority module. A compiler can check for violations within compilation units but run-time code would be needed for calls across separately compiled modules.

Limitations and alternatives

The significant limitation in the direct use of IOTRANSFER is that user processes are forced to employ busy waiting. This is acceptable if there is only one user process but is intolerable otherwise. If anyone did want to mix the direct use of IOTRANSFER with the standard module Processes then a safety precaution might be to give its implementation an explicit priority of zero. In theory this should trap any attempts to use signal operations from a device process but your system probably does not employ the necessary checks. This fanciful suggestion could also be tried with interface modules, designed for use with Processes, in order to protect mutual exclusion.

It may seem that a way forward would be to extend the standard Processes module to allow for device processes. User processes could then wait for a signal sent by a device process and in the mean time other user processes would be scheduled. The ETH M2RT11 compiler is distributed with a module PROCESSSCHEDULER which attempts this (be warned there are bugs). Their definition module adds two further procedures:

PAGE 55

```
PROCEDURE SENDDOWN(VAR s: SIGNAL);
PROCEDURE DOIO(va: CARDINAL);
```

3oth regular and device processes are still started with StartProcess and so are all linked into the ring together. The user is required to differentiate between them by using SENDDOWN and DOIO only in device processes. SENDDOWN simply marks the receiving regular process as ready. If the device process has interrupted another process then it returns to it directly on a subsequent WAIT or DOIO. If no ready process can be found on the ring then LISTEN is called before restarting the search - on the assumption that at least one regular process is waiting for a signal to be sent down by a device process which is in turn waiting for an interrupt

But all this solution does is give us a poor emulation of Modula-1. It remains the case that 'broadcast' signals are used for interaction with device processes even though the examples with IOTRANSFER have shown us that a closer coupling may be more appropriate. We also have the various restrictions on signal exchange between device processes and on waiting for interrupts in regular processes, although these can no longer be effectively policed. An alternative is to look for a new layer in our design - between the low-level coroutine mechanism and the higher level of signals. Then we have a chance of improving on Modula-1.

A new Processes module

The intermediate level module described here is based on an initial implementation for the PDP11 which has since been moved to the Motorola 68000 and the Intel 8086. It is used in the experimental ModOS real-time operating system which is being built with partial support from the UK ESRC.

The name *Processes* is kept and a higher level module offering synchronization primitives is given the more appropriate name of *Signals*. The definition module is as shown:

```
DEFINITION MODULE Processes;
(* © Roger Henry (Nottingham University) *)
(* IMPORT and EXPORT lists omitted for brevity *)

 TYPE Process;
 PROCEDURE NewProcess(
    code: PROC;
    priority: INTEGER;
    wkspAt: SYSTEM.ADDRESS;
    wkspSize: CARDINAL;
    param: CARDINAL;
    VAR p: Process);
 PROCEDURE Enable(other: Process);
 PROCEDURE SuspendMe;
 PROCEDURE SuspendUntilInterrupt(va: SYSTEM.ADDRESS);
 PROCEDURE Disable(other: Process);
 PROCEDURE Cp(): Process;
 PROCEDURE PriorityOf(p: Process): INTEGER;
 PROCEDURE MyParam(): CARDINAL;
 PROCEDURE MinWksp(): CARDINAL;
END Processes.
```

The low-level mechanism of independent coroutines and explicit transfers is replaced by a scheme in which processes are scheduled automatically according to simple priority rules. There is a fixed integer priority associated with each process. The main program becomes the initial (main) process and runs at a priority of 0. Additional processes may be created dynamically, during the execution of any process, by calling **NewProcess** and specifying the required priority. The returned value of type **Process** can be stored and remembered to identify the process in subsequent operations. The function **Cp()** will deliver this value for the current process. It is guaranteed that tests of equality and assignment of these **Process** values will produce sensible results (although the values might be recycled).

Transfers of control between processes occur in three cases. The first of these is when the current process enables another process by calling **Enable(p)** and the condition holds that **PriorityOf(p) > PriorityOf(Cp())**.

Secondly the current process may call **SuspendMe**. Then control passes to the process of highest priority that has been enabled for longest at that priority. An idle process is provided that is always enabled and has the priority MIN(INTEGER). The suspended process must have made arrangements to be enabled again if it is ever to regain control. This usually happens when an awaited event occurs.

The third case is when the current process suspends itself by a call of **SuspendUntilInterrupt**. The immediate effect is the same as for normal suspension. When the specified hardware interrupt arrives, the suspended process will be enabled automatically. It will resume execution as soon as the priority rules allow

This definition implies, correctly, that on this scheme any process may synchronize with an external interrupt. (But of course only one process at a time for each interrupt.) This can lead to much simplification in avoiding the need for special device processes enclosed entirely in a high-(processor)priority module. It turns out that these are only strictly necessary when the device generates interrupts spontaneously - as with keyboards and clocks.

MyParam() delivers the value given for **param** when the current process was started. This service is provided since the copying of global variables into local process variables can be unsafe. The reason is that when a process creates a new process at the same or a lower priority as itself and then enables it for the first time, there will not be an immediate transfer of control. The passed parameter is sufficient for direct use in many cases. In others, it can be used as an index into an array of per-process variables of any required type. The value 0 is returned in the main process.

Finally, before an example, the procedure **Disable** can be used to disable another enabled process. **Disable(Cp())** has no effect. Whenever it is known that a process will not be enabled again then the corresponding workspace may be reused.

Typical direct use for device handling closely follows the pattern for the direct use of IOTRANSFER. This is illustrated by suitably adapted versions of the procedures from Example 3 for buffered keyboard input. Now **user** and **driver** are variables of type **Process**.

```
PROCEDURE Print(ch: CHAR);
BEGIN
  IF n - N THEN
    user :- Cp(); userWaiting :- TRUE; SuspendMe;
  END;
  buff[in] :- ch; in :- in MOD N +1; INC(n);
  IF driverWaiting THEN
    driverWaiting :- FALSE; Enable(driver)
  END;
END Print;

PROCEDURE Handler;
BEGIN
  INCL(statusReg, intEnable);
  LOOP
    IF n - 0 THEN
      driverWaiting :- TRUE; SuspendMe
    END;
    outBuf :- buff[out]; out :- out MOD N +1; DEC(n);
    IF userWaiting THEN
      userWaiting :- FALSE; Enable(user)
    END;
    SuspendUntilInterrupt(intVec);
  END
END Handler;
```

The enclosing module must still specify a processor priority of 4. By convention, the driver process is started with a software priority of 40 = 4x10

```
NewProcess(Handler, 40, ADR(wsp), SIZE(wsp), 1, driver);
Enable(driver);
```

Through the parameter mechanism, it is possible to have several device processes executing the same procedure and serving multiple instances of a device where each has its own vector and register set.

The next example shows how the new operations are used without a special device process to achieve unbuffered output to a terminal without busy waiting.

```
MODULE Printer [4];
(* IMPORTs, EXPORTs and declarations omitted for brevity *)

  PROCEDURE Print(ch: CHAR);
  BEGIN
    IF NOT (readyBit IN statusReg) THEN
      INCL(statusReg, intEnable);
      SuspendUntilInterrupt(intVec);
      EXCL(statusReg, intEnable);
    END;
    outBuf  = ch
  END Print;
END Printer
```

The ability to set up vectors dynamically can now be used to great advantage.

PAGE 58

The scheduling strategy

Enabling a higher priority process than yourself is like asking the other process to interrupt you. An interrupted process remains enabled and so will remain the process that has been enabled for longest at its priority. When the interrupting process suspends itself, control will therefore return to the interrupted process before any other of the same priority. Of course, the interrupting process may have enabled some processes of intermediate priority and they will be executed first.

By design, an interrupting process cannot tell the identity of the process it has interrupted. There is therefore no way that the basic scheduling strategy can be subverted, say by a clock process which disables the interrupted user process and then selects another one which it has previously disabled.

A user may easily provide her or his own idle process by giving it a negative priority. For simulations of consumption and production activity, we have built a module employing such a low priority process. The exported procedure **Next**, stores the Process value of the caller in a free slot in a table and then calls **SuspendMe**. The idle process then selects a suspended process at random and enables it. It would be a simple matter to weight the probability of selection by the priority of the process.

Mutual exclusion

Given the usual arguments about not calling external routines which could cause a process switch, mutual exclusion in modules used as interfaces between processes is guaranteed - provided all calling processes are of the same priority. If more general interfaces are required then the interface module must specify a processor priority high enough to prevent all interrupts. No process switching can then occur other than that requested by the interface procedures. The value of the necessary priority will vary from machine to machine and so is exported from a separate module as the constant **monitorPriority**. It is kept separate so that it can be used with higher-level operations which would otherwise not require the user to import directly from **Processes**.

This monitor solution is extremely simple. Or rather it is simple but extreme. More selective exclusion can be provided by the use of semaphores. Operations on general semaphores are provided by a separate module which is implemented above the **Processes** layer.

Higher-level modules

At the **Processes** level, there is a deliberate separation of the creation and the enabling operations. The user must also provide the workspace. This choice is appropriate at this relatively low layer of a system since it avoids dependency on a storage allocator. A module at a higher-level exports the procedure **StartProcess** which adds an increment to what the caller thinks is needed for the amount of workspace (using **Processes.MinWksp()**), allocates it from a heap, and then creates and enables the process. The current implementations use the given procedure directly as the program of the process but it is likely that future versions will arrange for it to be called indirectly. This will allow processes to terminate cleanly as far as the client is concerned. Already it is possible to call **StopMe** to terminate and arrange for the workspace to be released to the heap. The killing off of one process by another is not envisaged since it would assume knowledge of data structures built by other modules, from which references to the dying process would need to be removed.

Other modules built on top of **Processes** which build such data structures include **Signals**, **Semaphores** and **Timing**. A signal queue now becomes simply a linked list of records containing the **Process** values of the waiting processes. There is no need to have a record field allocated for the queue in the process descriptor itself. This is part of the layered approach to system design.

The consistent interpretation of the priority rules forces a change in the semantics of the signal operations between processes of equal priority. Now when the first waiting process is enabled, there will not necessarily be an immediate transfer of control. (However, the waiting process will have been enabled longer than any process subsequently receiving a signal and so will start to execute first.) The moral is that interface procedures using signals for synchronization, should be written to make no assumptions about the relative speeds (scheduling) of two ready processes (the receiver and sender of the signal). Rewriting the semaphore operations of Example 1 accordingly we have

```
PROCEDURE P(VAR s: Semaphore);
BEGIN
  IF s.taken THEN
    Wait(s.free)
  ELSE
    s.taken := TRUE
  END
END P;

PROCEDURE V(VAR s: Semaphore);
BEGIN
  IF Awaited(s.free) THEN
    Send(s.free)
  ELSE
    s.taken := FALSE
  END
END;
```

It is now impossible for the signalled condition to be invalidated before the waiting process proceeds. (The general **Semaphores** module of the ModOS system uses the **Processes** operations directly.)

The time at which internal and external events occur is of great importance in real-time laboratory applications. The **Timing** module allows the time of output events to be controlled and the time of input events to be measured. The essential idea is to allow all user processes to delay for a specified period by sharing the use of a single hardware interval timer. The clock is set to interrupt at the end of the period which will expire first. A high priority device process handles the interrupt and enables the corresponding user process before restarting the clock for the next interval. The **Signals** module makes use of **Timing** to offer timeouts during waits.

Conclusion

I hope that you are now convinced, as I have been, that Modula-2 is of considerable value in building portable real-time systems in a layered fashion. That the end result is worthwhile is demonstrated by the efficiency which can be achieved without sacrificing structured design. As an indication of this, we can achieve millisecond precision in timing using a DEC LSI-11/23.

Modula-2 News # 0 October 1984

Revisions ... to Modula-2, Wirth
Spec. of Standard Modules, Hoppe
Modula-2 bibliography, Brown
Modus Membership list
Modula-2 Implementation Questionaire

Modula-2 News # 1 January 1985

Letter to Editor, Layman
Letter to Editor, Bush
Gleaves' Modula-2 text, DeMarco
MODUS Paris meeting, Blunsdon
Report of M2 Working Group, Souter
Library Rationale by Randy Bush
Library Definition Modules
Library Documentation by Jon Bondy
Validation of Modula-2 Impl, Siegel

MODUS Quarterly # 2 April 1985

Letter on Library, Anderson
Letter to Editor, Emerson
Comments on Modula-2, Emerson
Opaque Types, French & Mitchell
Dynamic Instantiation, Sumner
Linking Modula-2, Symons
Library Comments, Peterson
Modula Compilers, Smith
Coding War Games, DeMarco
M2, Alt. to C, Djavaheri/Osborne

MODUS Quarterly # 3 July 1985

Letter re opaque types, Endicott
Letter on language issues, Hoffman
Modula-2 in "Real Time", Barrow
RajaInOut: safer I/O, Thiagarajan
Contentious Problems, Cornelius
Expressions in Modula-2, Wichmann
Scope Problems: Modules, Cornelius
Corrections to compiler list

MODUS Quarterly # 4 November 1985

MODUS Meeting Report by Bob Peterson
A Writer's View of a Programmer's Conference by Sam'l Bassett
Concerns of a Programmer by Dennis Cohen
A Few Modifications to a Modula Standard Library Proposal by Nagler & Sieg
Proposal for a standard library and an Extension to Modula-2
by Martin Odersky, Peter Sollich & Mike Weisert
Standard Library for the Unix OS by Morris Djavaheri
An Implementation of the Standard Library for PC's by E. Verhulst
Modula-2 Compilation and Beyond by D.G. Foster
Modula-2 Processes - Problems and Suggestions by Roger Henry

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

Hints for contributers: (If in doubt, send it in; we can then discuss i

Send CAMERA READY copy to an editor (dot matrix copy is usually unacceptab Machine readable copy is preferred. Present facilities permit printing pa of 60 lines (80 characters wide). Pages limited to 70 characters per line be printed in 10 pitch (Pica). Page numbers will be shown on line 60 if s permits. Long or loosely formatted contributions may be reduced for print

Your text will not be altered without your permission. Editorial remarks be enclosed in square brackets [ ]. Additional text may be added to aid th reader in identifying the subject and author of contributions. Modula-2 N is copyrighted, but the author retains all rights to further publication o contributed material. Working papers and notes about work in progress are encouraged. Modula-2 News is not perfect, it is current.

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

Richard Karpinski	TeleMail	M2News or RKarpinski
6521 Raymond Street	BITNET	Dick@ucsfcca
Oakland, CA 94609	Compuserve	70215,1277
(415) 666-4529 (12-7 pm)	UUCP	...!ucbvax!ucsfcgl!cca.ucsf!dick
(415) 658-3797 (ans. mach.)		

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : ____

Affiliation : ____

Address : ____

Address : ____

State : ______________ **Postal Code** : ________ **Country** : ______________

Phone : (____) ____ - __________ **Electronic Addr** : ______________

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

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

Implementation(s) Used ______________________________________

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

P.O. Box 51778
Palo Alto, California 94303
United States

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

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

The Modula-2 Users' Association is a forum for all parties interested in the Modula-2 Language to meet 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 will be information on the status of the language, along with examples and ideas for programming in Modula-2. For everyone, there is information on current known implementations and other resources available for information on the language.

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : ____________________

Affiliation : ____________________

Address : ____________________

Address : ____________________

State : __________ **Postal Code** : ______ **Country** : __________

Phone : (____) ____ - ________ **Electronic Addr** : __________

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

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

Implementation(s) Used : ____________________

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

P.O. Box 51778
Palo Alto, California 94303
United States

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

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

The Modula-2 Users' Association is a forum for all parties interested in the Modula-2 Language to meet 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 will be information on the status of the language, along with examples and ideas for programming in Modula-2. For everyone, there is information on current known implementations and other resources available for information on the language.

MODUS
c/o Pacific Systems
PO Box 51778
Palo Alto, CA 94303
USA

MODUS
Postfach 289
CH-8025 Zuerich
Switzerland

Return Postage Guaranteed