The MODUS Quarterly

Issue # 7

February 1987

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

CONTENT

Without Delay Renew Today!

Directors of MODUS, the Modula-2 Users Association:

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

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

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

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

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

Administration and membership:

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

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

Editor, Modula-2 News:

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

>> Problems? Missing an issue? <<

Contact your membership coordinator (see above).

Publisher:

George Symons (see above)

Putative publication schedule:

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

Submissions for publication:

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

TeleMail address: M2News

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Correspondence not for publication should be PROMINENTLY so marked.

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 : _____

Without Delay Renew Today!

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

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

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

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

Aline Sigrist
ERDIS SA
CH-1800 Vevey 2

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

Modula-2 News Issue # 0 October 1984

Purposes, practices and promises for Modula-2 News
Revisions and Amendments to Modula-2, Niklaus Wirth
Specification of Standard Modules, Jirka Hoppe
Modula-2 in the Public Eye (a bibliography), Winsor Brown
Modus Membership list, by name
Modus members's addresses, by location
Modula-2 Implementation Questionnaire

Modula-2 News Issue # 1 January 1985

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

MODUS Quarterly # 2 April 1985

Editorial
Letter on the draft Modula-2 Library, T. Anderson
Letter to the Editor, Mark Emerson
Opaque Types in Modula-2, C. French & R. Mitchell
Dynamic Module Instantiation, Roger Sumner
The Linking Process in Modula-2, Jeanette Symons
Modula-2 Library Comments, Bob Peterson
Modula Compilers - Where to Get 'em, Larry Smith
Coding War Games Prospectus, Tom DeMarco
M2, An Alternative to C, M. Djavaheri, S. Osborne

MODUS Quarterly # 3 July 1985

Editorial & potpourri of mail
Letter re opaque types, Steve Endicott
Letter on language issues, Christian Hoffman
Some Thoughts on Modula-2 in "Real Time", Paul Barrow
Letter re "actual Modula-2 code", Raja Thiagarajan
RajaInOut: simple, safer, I/O for Logitech/MS-DOS, R. Thiagarajan
Selection of Contentious Problems, Barry Cornelius
Expressions in Modula-2, Brian Wichmann
The Scope Problems Caused by Modules, Barry Cornelius
Corrections and additions to Modula-2 compiler list

MODUS Quarterly # 4 November 1985

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

MODUS Quarterly # 5 February 1986

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

MODUS Quarterly # 6 November 1986

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

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

Hints for contributors:

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

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

Send your submissions to:

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

Without Delay Renew Today!

Was that loud enough? Not only is it time to renew, but we will have a dynamite issue # 8. The main reason that the hot stuff in number 8 is not in this issue is that I'm avoiding any possible delay in getting this one out. It really helps to have an assistant editor providing hours of help every week! There were a few weeks across the holidays when nothing at all got done about this issue but then I never promised to give up my whole life for MODUS.

Stan Osborne is helping me to achieve something approaching a regular schedule. I offer him my thanks. Does anyone care to offer an opinion as to what we should do more of or less of? How can I serve you if I can't tell what you want or even how I'm doing? Be specific.

Stan is also in the early stages of planning for a MODUS meeting in the San Francisco area in the first half of June, 1987. This is, of course, quite tentative. Unless great obstacles overcome us, issue # 8 will follow this one within a few weeks. By that time, we should have rather better information about the June meeting.

This issue (# 7) has some actual Modula-2 code in it. The ability to add modules was specifically intended to provide a vehicle for all those things which might have been put into the language but were not. I invite you to observe what limitations are imposed by this style of adding a feature to Modula-2. Is this SET OF CHAR module enough that we need not press the standards group to require SET OF CHAR within the language proper? I would like to hear your answer and reasoning.

Speaking of standards, there is just a taste of what goes on in this issue, but issue # 8 will have some substantive issues presented. Another sense of standard is also in evidence in the current issue; Modula-2 is obviously in use for standard projects like implementing a version of TCP (Transmission Control Protocol) and class projects to build an operating system. The language is also showing up on standard computers (read IBM mainframes).

If you look closely, you will see that we have a new MODUS Secretary in Europe. Furthermore, several board members have new addresses or phone numbers. We really do try to keep things up to date and accurate.

Several people have asked for an address for Frode Odegard. He can be reached at Modula-2 CASE Systems A/S, Sands Veien 4B, N-2050 Jesheim, Norway or on BIX as Frode or via Fido 502/25 (private node).

Morris Djavaheri gave me a translation of the floating-point testing program PARANOIA into Modula-2. It is the fifth language for the program, as far as I know. Contact me if you want a copy in any language. For now, it is available also in BASIC, FORTRAN, Pascal, and C. There was an article about it in the Feb 85 Byte Magazine.

Dick Karpinski
Editor

New Products

Modula-2

For IBM 370 VM/CMS. Developed by the Computer Systems Group at the University of Waterloo. Contact Sandra Ward, WATCOM Products Inc., 415 Phillip Street, Waterloo, Ontario, Canada, N2L 3X2, (519) 886-370, Telex: 06-955458.

EXE2LNK

For users of the Logitech compiler on IBM PC machines. Converts .EXE files to .LNK files used by the Logitech Modula-2 Linker. Used to bind assembly language routines with Modula-2 programs. Contact Leif Ibsen, Blommevangen 15, DK-2760 Maalov, Denmark.

M23

Third Edition Modula-2 for RT11SJ, RT11XM, and TSX+., plus other performance improvements. Contact: Dr. K. John Gough, School of Computing Studies, Queensland Institute of Technology, G.P.O. Box 2434, Brisbane, Queensland, Australia, 4001, Telex: 44699.

Modula-2PC

Modula-2 for IBM PC/PCjr/PCXT/PCAT or compatible. Contact: Peter Collier, PCollier Systems Inc., Suite 390, 7925-A North Oracle Road, Tucson, Arizona 85704, (800) 522-2060.

MacMeth

Modula-2 for 512K Macintosh or Macintosh Plus. Contact: Modula Corporation, 950 North University Avenue, Provo, Utah 84604, (800) 545-4842.

GEFI Modula-2

An adaptation of the ETH-Zuerich SMILERX, 68000 Modula-2 compiler to the Macintosh environment. Contact: Chantal Fauconnet, GEFI Service, 71 rue de la Victoire, 75999 Paris France, Tel: (1) 39 85 44 43.

Modula-2/RTS

Modula-2 for RT11 and SHAREplus. Contact: Guenter Dotzel, ModulaWare GmbH, Wilhelmstrasse 17A, D-8520 Erlangen, West Germany, Tel: 09131 208395.

Modula-2 for OS/MVS

A fast single pass compiler for IBM Mainframes. Generates native 370 code for OS linker and loader. Contact: G. Blaschek, University of Linz, Institut fuer Informatik, Altenbergerstrasse 69, A-4040 Linz, Austria, Tel: (0732)-232381-447

Modula-2/68
Modula-2/68-CD

A Modula-2 language system for use with the MC68000 family of computers. Native Modula-2 development on many of the popular 68000 based systems. Many host systems are supported. CD is Modula-2/68 cross development on VAX computers. CD runs on VAX/VMS, Ultrix and BSD 4.2/4.3 operating systems. Contact: Stan Osborne, Djavaheri Bros., P.O. Box 4759, Foster City, California 94404-0759, Tel: (415) 341-1768, Telex: 4949940.

Modula-2 VM/CMS (IBM 370)

Developed at TU Berlin. Contact: Thomas Habernoll, TU Berlin, Informatik Rechnerbetrieb, Sekr. FR 5-3, Franklinstr. 28/29, 1000 Berlin 10, West Germany.

Modula-2

Modula-2 and Advanced Systems Editor from Pecan Software Systems, Inc. Contact: Y. A. Lifschutz, 1410 39th Street, Brooklyn, NY 11218, Tel: (718) 851-3100.

Modula-2 Standardisation: The go-betweens' tale

Jim Welsh and Paul Bailes

Department of Computer Science
University of Queensland
St Lucia
Queensland 4067

Background

This brief report arises from a visit by the authors to the UK and Switzerland in June 1986. This visit was undertaken as part of an evaluation of the current state of Modula-2 for teaching purposes, the evaluation being carried out on behalf of, and with funding from, the Australian Commonwealth Scientific and Industrial Research Organisation.

During our visit to the UK we attended a day-long meeting of the BSI working group on standardisation of Modula-2, having previously had discussions with two members of that group, namely Brian Wichmann at NPL, and Derek Andrews at the University of Leicester. In Switzerland we spent three days at ETH which included discussions with Niklaus Wirth on standardisation issues. The purpose of this report is to compare the positions taken by the BSI group on the one hand, and Niklaus Wirth on the other, on language problems currently identified by BSI.

In subsequent sections we try to summarise the outcome of our discussion of particular issues with NW. In general each section consists of a statement of the problem as we expressed it to NW, a summary of NW's reaction, and in some cases a summary of our own view in the light of these reactions.

In providing this report we must emphasise its limitations through our own inadequacies, either in expressing the problem as perceived by the BSI group or in capturing NW's reaction. For this reason we strongly recommend more direct communication on critical issues. In our discussions NW made it clear that he welcomes the development of a Modula-2 standard. He is not interested in direct participation in the time-consuming process required to develop such a standard, but is willing to respond to specific issues put to him by BSI. In the light of the lack of communication to date, and the inadequacy of such transient intermediaries as ourselves, we strongly recommend that BSI pursues this option in some appropriate manner.

Philosophies of language standardisation

Before describing NW's reaction to particular issues it is worthwhile to summarise his view of what a programming language standard should be, since that view diverges somewhat from that supported by BSI.

The BSI group aims to produce a Modula-2 standard which is similar to the Pascal standard in that it resolves program behaviour across the complete spectrum of possible usage of the language features, whether or not these are in the spirit of the language design, or represent good programming practice. For Modula-2 this semantic completeness of the standard will be further reinforced

by the development of a formal VDM semantic model in which only those language aspects that are clearly implementation-dependent will be left undefined.

NW is not in favour of a VDM model of semantics, precisely because of the semantic completeness which it tends to enforce. He believes instead that a language standard should only guarantee program behaviour where the language features are used in a way that is consistent with the overall design intent and good programming practice, leaving other usage in some undefined or unstandardised state.

This attitude is based on the view that good software is only achieved by programming practice which consciously avoids the use of language features in obscure, unusual or marginal ways. He therefore feels that a standard which labels such usage as unreliable or unsafe is more effective in promoting good software than one which forces implementors to conform to particular (and sometimes arbitrary) semantic interpretations of such usage, and thereby encourages such use.

In applying this general view to the development of a standard for Modula-2, NW's attitude on many issues might be characterised by the maxim *When in doubt, leave it out!* (the WIDLIO principle). On those issues which seem contentious, either through variation between documents or implementations, or through lack of consensus on the value of a given feature, his first reaction is to question whether the language features concerned should be part of the standard at all.

While we have sympathy with NW's view that a language standard should promote good programming practice, a standard of the sort proposed seems an even more difficult goal to achieve by means of a standardisation committee, and to defend or police thereafter, since it depends on more subjective judgements of what is good! It seems inevitable to us, therefore, that the BSI group must pursue semantic completeness on those features included in the standard. They might, however, give serious thought to the option of leaving certain features out completely, as this strategy may not pose additional problems to the same extent.

Export clauses in Definition Modules

One change between the Modula-2 language as defined by the last two editions of *Programming in Modula-2* is the elimination of the need for an EXPORT clause in a definition module - by adopting the convention that all objects mentioned defined in the definition module are automatically exported. The BSI group see this as a retrograde step, in that it reduces the control over what can be accessed by client modules of the module so defined. NW on the other hand sees the change as a rationalisation consistent with the basic purpose of definition modules.

This divergence of views seems to reflect a difference in emphasis on the two perceived functions of definition modules, which are

(1) to enable separate compilation;

(2) to capture the abstraction intended by a module design.

As developments in languages such as Ada demonstrate, abstraction makes it desirable to preclude external use of some attributes of a server module to which a compiler must have access in compiling its clients. However, as the features provided for this purpose in Ada also show, achieving the precise interface control required by abstraction is a non-trivial language design problem, and Modula-2's limitations in this area are generally recognised.

As we understand it, the BSI group have rightly taken the decision not to attempt to "improve" Modula-2's abstraction control by introduction of new features. In this sense their attitude on export from definition modules seems somewhat inconsistent. Having recognised Modula-2's limitations with respect to precise abstraction control, the dropping of explicit export from definition modules seems a logical choice.

Export Semantics

Derek Andrews raised a number of issues within the BSI group with respect to the details of export semantics.

One such issue is whether all attributes of an object are exported with its identifier. Suppose, for example, a module exports a variable of a local type which either is anonymous or is named but not itself exported. Has the importing module knowledge of the structure of, and operations applicable to, the variable?

NW believes that such attributes must be automatically exported. This view again seems consistent with the choice between meeting the needs of a compiler and those of abstraction control discussed in the previous section.

The alternative within an identifier-based export system seems to involve the naming and exporting of all intermediate types involved in the description of any object whose attributes are to be made available. This implies considerable clumsiness in many simple situations.

Furthermore, the language already has a specific mechanism for attribute hiding (opaque types), albeit with unpleasant restrictions imposed by separate compilation.

For these two reasons automatic export of attributes seems to us the logical choice.

Other export issues raised by Derek Andrews were not discussed with any positive result.

The type CARDINAL

During our visit the BSI group reviewed the issue of the types INTEGER and CARDINAL. The group had previously decided to maintain Modula-2's original conventions (that INTEGER and CARDINAL are distinct types with distinct arithmetics, but with special assignment compatibility rules between them). They were therefore dismayed to find from documentation of a recently developed ETH compiler that CARDINAL had been redefined as a subrange of integer by that implementation. In the light of this apparent change the group reviewed its previous decision, but in the end decided to adopt an even stricter definition - of INTEGER and CARDINAL as distinct types but without special assignment compatibility rules!

NW was concerned to hear that the group's decision was triggered by an ETH document that was not meant for external distribution, and certainly not meant to be taken as a general redefinition of Modula-2.

In discussing the issue itself NW made clear that

(1) the original definition of CARDINAL as a distinct type with distinct arithmetic was dictated by address calculation requirements on 16-bit machines;

(2) he no longer considers such requirements relevant to future definitions of the language.

In keeping with the WIDLIO principle, his first choice would be to exclude CARDINAL from the language standard, with corresponding redefinition of standard functions, etc. If, however, the need to preserve the standard identifier CARDINAL is seen to preclude this choice, he would favour its definition as a subrange of INTEGER, such that most existing programs would continue to work, albeit with some redundant type coercions within them.

Concurrency

The BSI group has identified the issue of concurrent programming as a significant troublespot in Modula-2. The subgroup delegated to consider options in this area has not yet reported, but the intention seems to be that some explicit features for concurrency must be defined.

In contrast NW strongly adheres to the WIDLIO principle on this issue. He recognises that the features originally defined in Modula-2 were influenced by needs of a single-processor machine with a particular style of device control. Furthermore he believes that no consensus on a general machine-independent model for concurrency yet exists, or indeed can be expected to at this stage. He therefore feels that the language standard should not prescribe any such model, but should leave implementations to provide appropriate facilities via suitable library modules.

Modula-2 for VM/CMS (IBM/370)

Here is some information about
- the history, present and future of our project
- the actual implementation
- and how to get it.

The History of Modula-2/CMS

It should be superfluous to enumerate reasons why we are interested in a Modula-2 implementation for /370 machines. But how to get it? For a while we tried the strategy "wait until someone has done the work and use Pascal/VS in the meantime". We tried this very hard but it did not work.

Phase -1 (once upon a time)

One of our students (Thomas Kilian) came to me and was willing to do a lot of work for no money (a rare species). Fortunately we found what he was searching for. He was impressed enough by Modula-2 to ignore all difficulties. At this time we had only a listing and some documentation of the Smiler-2 system (written in CD-Pascal). Our idea was to adapt it to Pascal/VS and add a code generator for /370 code. Thomas Kilian started - first with the work and then with curses: the CD-Smiler uses some CD-Pascal specific features. I don't know whether it is a good tool to work with, but it is not a good base for porting. So after some frustration we came to

Phase 0 (zero)

This is the right name for what occurred. Thomas Kilian stopped working for nothing and started a job (likewise frustrating but he was well paid). Because more and more implementations of Modula-2 were coming up, we thought it should be the right time to apply our first strategy (wait until ...). There were some rumours about ongoing implementations for /370 machines. However rumours are not executable - even on a virtual machine. And so we came to

Phase 1 (something is going on)

In the early summer of 1985 Thomas Kilian came back. In the meantime we had Modula-2 running on some Unix machines (Arnfried Ossen (OSSEN@DB0TUI11.BITNET), who posted the first message on Modula/CMS on the net, had invested some effort to adapt the Cambridge Motorola 68K implementation on System V, BSD 4.?, and Unos machines). We considered this Modula written system to be a better base for our work than the CD Pascal Smiler. There were two options:

1) Development of a code generator for /370 code in Modula-2 on a Unix machine, cross-compilation of the entire system.

2) Rewriting of the compiler front end and implementation of the code generator in Pascal/VS under VM.

The first way seems to be shorter. We chose the second way. The overhead seems to be the rewriting of the compiler front end. This task consists of two parts: translation from Modula towards Pascal/VS (which is a mechanical procedure) and adaption of the operating and file system dependent parts (had to be done even if we had chosen the first way). The problem of the first way would be the test phase (compile under Unix -> file transfer to VM -> test -> crash -> back to Unix - and so on ...).

Here we are. The front end (which generates M-code) and an interpreter are running under VM/CMS. We now enter

Phase 2 (the future has just begun)

Actually we are able to compile (sufficiently fast) Modula-2 programs and execute them (insufficiently slow). The interpreter is good enough to allow playing around with Modula and gaining experiences.

Last Christmas I sent a message to Santa Claus that I wish for a fast tool to do system programming including i/o and interrupt handling. Apparently his mailer was down, so we started with the code generator. Maybe we shall have sufficient results in the middle of this year.

What does 'sufficient' mean within this context?

- Execution speed. The first release should not be very much slower than Pascal/VS. There are some optimizations planned (and not yet implemented). Nevertheless it is more important to have a working code generator than a fast collection of bugs.

- Even though conflicts may arise with execution speed, we have decided to make the code relocatable. This is necessary for fast loading of CMS nucleus extensions and should be useful for some other applications.

- Because I don't trust HNDINT, HNDEXT, and STAX macro processing (it is my historical prejudice), the run time environment should allow Modula programs full control of interrupts. Even if you trust HNDINT it is not efficient to let CMS look at thousands of interrupts, which are to be handled by your own code. Maybe there should be two variants of the run time environment, one for normal applications and one - quick and dirty - for hackers who hate assembler.

Phase 3 (in the year 2525)

Your Modula program has solved some great problem and writes the result (the value 42) formatted and/or binary to a file and reads the next question from another file. You don't trust in your program and call the debugger, which uses the full screen 3270 i/o package. You find the bug and correct it using the wonderfull full screen syntax directed editor. Wouldn't it be nice?

The Modula-2/CMS Compiler

To avoid reinventing all the wheels of our vehicle we concluded to base our work on an existing compiler (a version running under Unix). The translation from Modula to Pascal/VS is mostly a mechanical act. Because it is frustrating to do this by hand, Thomas Kilian wrote a 'small' program. It accepts as input a subset of Modula (as used by the compiler) and generates an anotated Pascal/VS program. Only a small part was to be translated by hand (supported by the generated anotations), e.g. the string handling.

A little bit more work was to be done for file i/o and other operating system dependent stuff. This was mainly due to a mistake. I transferred the source files of the compiler from a Unix system to VM but I forgot the sources of the library. Thomas Kilian didn't know that there were sources of the library and had to guess the meaning of the calls. He did it very well and implemented a library system for code and symbol table informations. So the number of files on your disk will be reduced. Another problem solved by the libraries are the mapping of file names to module names. So module names are not restricted to 8 significant characters as CMS file names are.

If you don't like the error diagnostics and recovery of the Unix Modula-2 compiler you will be disappointed by Modula/CMS. It is the same mess.

The first test set of the compiler consists of the original Modula sources of the Unix version. The compiler came through without crash. This is nice but not a sufficient indication for the correctness of the generated M-code. But how to test it? Without further processing it is not possible to test the code under Unix (mainly because it's a mixture of EBCDIC strings and binary code values).

Even if the code is correct, what is its meaning? You should know it when you have to write a code generator. And so we step to

The M-Code Interpreter

All what we had was a sample interpreter, a sort of functional description of the Lilith machine. Because sometimes things are better understandable when you touch them, Thomas Kilian converted it to Pascal/VS. He had no ambitions to make more than a tool to understand and check out M-code sequences.

But better slow Modula than no Modula. While Thomas Kilian started with the code generator, I tried to make the interpreter usable, i.e. to add some i/o facilities and to improve speed. Although I hate benchmark tests (tell me your result and I give you the benchmark test that fits), I took a simple program (Ackermann, CPU bound, highly recursive) to get a feeling how slow the interpreter was. Then I started with tuning. The price: decreased readability of the interpreter. Although 7 times faster than the first version the beast was too slow.

Better slow Modula than very slow Modula. Writing Assembler to get Modula -- terrific. Actually only a part of the M-code instructions is realized in Assembler. The slow (Pascal) interpreter tries to switch to the fast (assembler) interpreter when possible. The fast interpreter is running until it fetches a non-implemented instruction. Then it gives control back to the slow but complete interpreter. So execution times of Modula are 6 - 100 times slower, compared with corresponding Pascal/VS programs.

Only a few library modules are implemented: simple terminal i/o, time of day, access to program parameters, a mapping of Pascal file i/o (untested).

Distribution of Modula/CMS

Damned networks. Nobody would know anything about the compiler. We had planned to howl around when the code generator is running. The interpreter was considered as a working tool to crack the meaning of the M-code instructions.

The first sentence isn't true. I like notworks, sorry, networks. But you have been warned. The support is lousy, the documentation is lousy, it's just like buying software from Microsoft. Please let me know whether you wish to

- get the current product (slow and ugly) as soon as possible
- wait a month to get the current product (a little bit polished and a little bit faster)
- wait for the first working version of the code generator (if I were a software vendor I would say: Real Soon Now).

Actually we have no time to copy tapes and ship them. All we can do are some key strokes to send the software via EARN/BITNET (and it is faster too).

86/01/28 Thomas Habernoll <HABERNOL@DB0TUI11.BITNET>

P.S. Does anybody use Reals? (You know Reals are these funny things that you use in Pascal/VS to force doubleword alignment for records containing CCWs). Sorry, please use Pascal, Fortran, or Basic. Reals are not yet implemented!

Thomas Habernoll
TU Berlin
Informatik Rechnerbetrieb
Sekr. FR 5-3
Franklinstr. 28/29
1000 Berlin 10
West Germany

Transmission Control Protocol (TCP) Implementation in Modula - 2

Fanyuan Ma and Larry D. Wittie

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, N.Y. 11794

ABSTRACT

The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication network and in interconnected systems of such networks. This paper describes the functions to be performed by TCP, the Tcp module that implements TCP in Modula - 2, and user commands in the MICROS/SAM2S operating system for transmitting packets.

1. Overview of TCP

Beginning with ARPANET, the Defense Advanced Research Projects Agency (DARPA) has sponsored the development of the Internet Protocol suite[1]. One of the prime uses for computer communication networks is to reliably transmit and receive files and electronic mail. The characteristics of these applications require passing a fairly large amount of data reliably and reconstructing the data in sequence. To support such Internet services, the Transmission Control Protocol (TCP) was developed.

TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications[2]. TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit-switched networks.

1.1. Interface

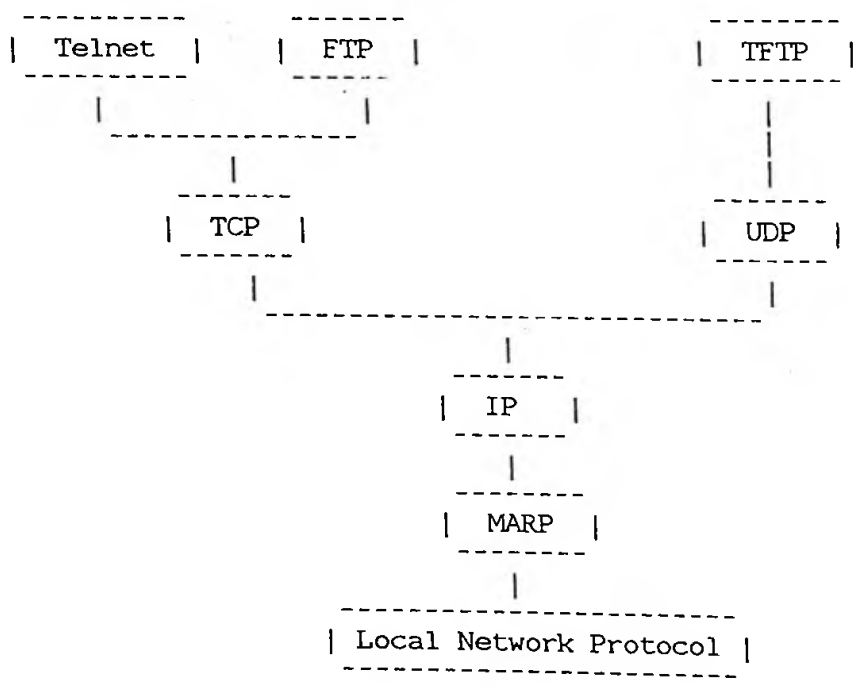


Figure 1: Protocol Relationships

TCP interfaces on one side to user or application processes and on the other side to a lower level protocol such as the Internet Protocol. Figure 1 illustrates the place of TCP in the protocol hierarchy.

The interface between an application process and TCP consists of a set of calls much like the calls an operating system provides to an application process for manipulating files. For example, there are calls to open and close connections and to send and receive data on established connections. It is also expected that TCP can asynchronously communicate with application programs.

The interface between TCP and lower level protocol is essentially unspecified except that it is assumed there is a mechanism whereby the two levels can asynchronously pass information to

each other. TCP is designed to work in a very general environment of interconnected networks. The lower level protocol which is assumed throughout this document is the Internet Protocol.

1.2. Operation

The primary purpose of TCP is to provide reliable logical circuit or connection service between pairs of processes. It contains mechanisms to provide reliable transmission of data. These mechanisms include basic data transfer, reliability, flow control multiplexing, connections, precedence and security. The basic operation of TCP in each of these mechanisms is described in the following paragraphs.

1.2.1. Basic Data Transfer

TCP is able to transfer a continuous stream of octets (bytes) in each direction between its users by packaging some number of octets into segments for transmission through the internet system. In general, TCP modules block and forward data at their own convenience.

1.2.2. Reliability

TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted.

At the receiver, sequence numbers are used to order segments correctly that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.

1.2.3. Flow Control

TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission.

1.2.4. Multiplexing

To allow for many processes within a single host to use TCP communication facilities simultaneously, TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets may simultaneously be used in multiple connections between process pairs.

1.2.5. Connections

When two processes wish to communicate, their TCP modules must first establish a connection by initializing status information on each side. When their communication is completed, the connection is terminated or closed to free buffers, table entries and other resources for other uses. Since connections must be established between unreliable hosts and over the unreliable internet communication system, a handshake mechanism with clock-based sequence numbers is used to avoid erroneous initialization of connections.

1.2.6. Precedence and Security

The users of TCP may indicate the security level and priority of their communication. Provision is made for default values to be used when these features are not needed. In our implementation, we do not define operations for these features.

2. Functional Specification

This section specifies the formats for the TCP header and transmission control blocks. It shows the sequence of control states for TCP and shows how a TCP connection for data is opened, used, and closed.

2.1. TCP Header Format

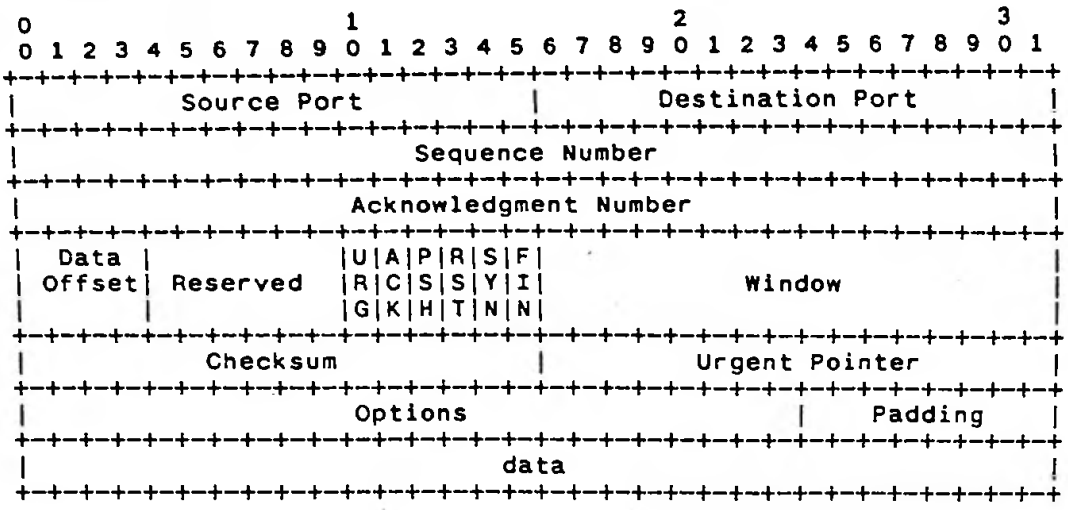


Figure 2: TCP Header Format

The Internet Protocol (IP) header carries several information fields, including the source and destination host addresses. As defined in Figure 2, a TCP header follows the internet header, supplying information specific to the TCP protocol. Figure 3 shows the pseudo header. This information actually is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by TCP on IP.

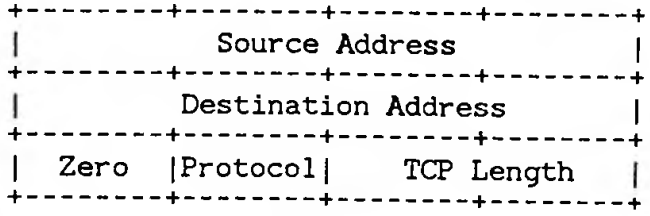


Figure 3: Pseudo Header

2.2. Transmission Control Block

Maintenance of a TCP connection requires remembering several variables. A connection record called a Transmission Control Block (TCB) is designed to store such variables as local and remote socket numbers; the security and precedence of the connection; and pointers to the user's send and receive buffers, the retransmit queue, and the current segment. In addition, several variables relating to the send and receive sequence numbers are stored in the TCB.

Send Sequence Variables:

SND.UNA:	send unacknowledged.
SND.NXT:	send next.
SND.WND:	send window index.
SND.UP:	send urgent pointer.
SND.WL1:	segment sequence number used for last window update.
SND.WL2:	segment acknowledgment number used for last window update.
ISS:	initial send sequence number.

Receive Sequence Variables:

RCV.NXT:	receive next index.
RCV.WND:	receive window index.
RCV.UP:	receive urgent pointer.
IRS:	initial receive sequence number.

2.3. Connection State

Figure 4 illustrates connection state changes, together with causal events and resulting actions. A connection progresses through a series of states during its lifetime.

The states are as follows:

LISTEN: represents waiting for a connection request from any remote TCP and port.

SYN-SENT: represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED: represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED: represents an open connection, on which data received can be delivered to the user. This state is the normal state for the data transfer phase of the connection.

FIN-WAIT-1: represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2: represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT: represents waiting for a connection termination request from the local user.

CLOSING: represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK: represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP.

TIME-WAIT: represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

CLOSED: represents no connection state at all.

A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS; the incoming segments, particularly those containing the SYN, ACK, RST and FIN flags; and timeouts.

The following sections explain the causing events and resulting actions briefly.

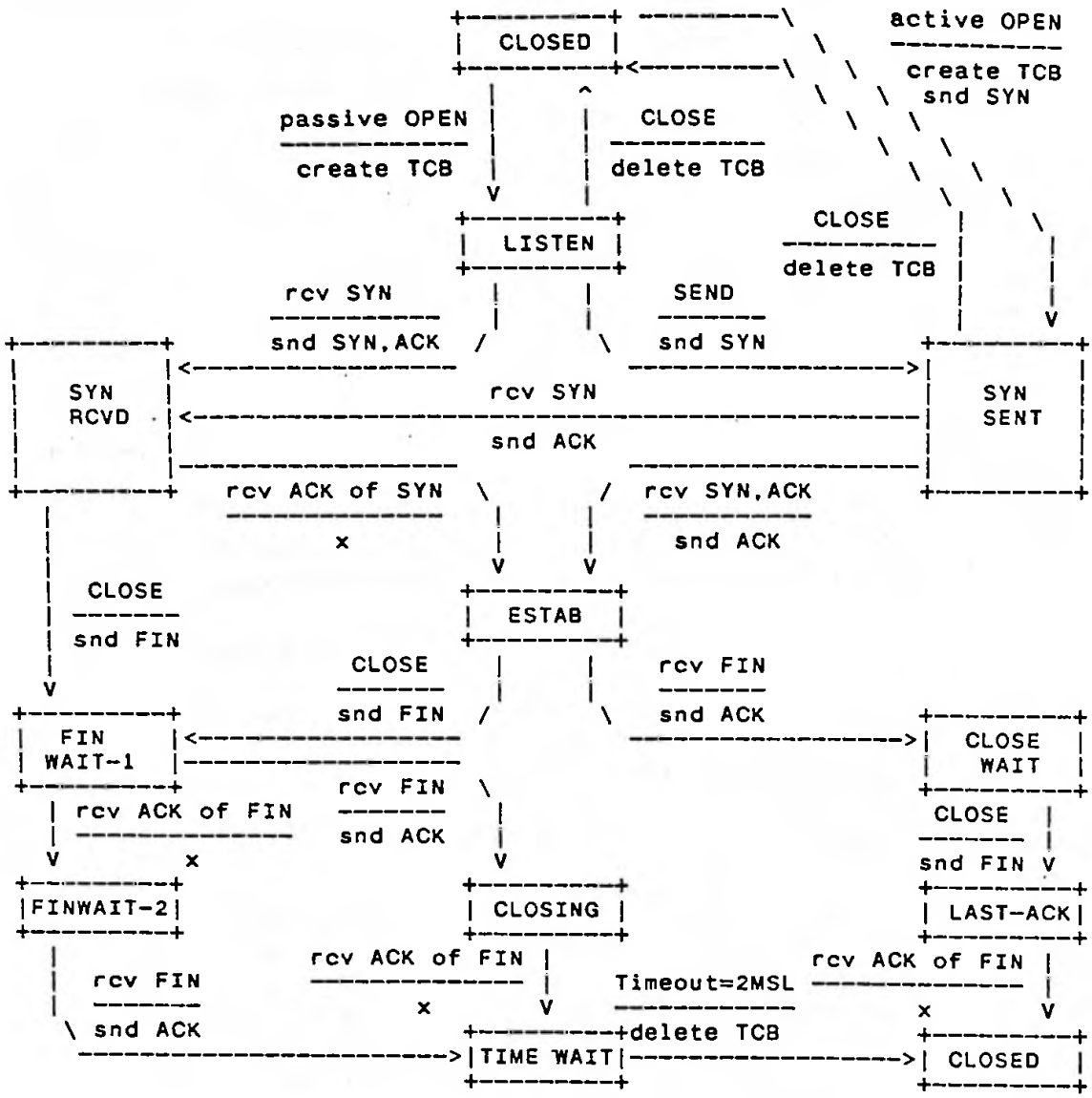


Figure 4: TCP Connection State Diagram

2.4. Sequence Numbers

A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. Since every octet is sequenced, each of them can be acknowledged.

The acknowledgment mechanism employed is cumulative so that an acknowledgment of sequence number x indicates that all octets up to but not including x have been received. This mechanism allows for straight-forward duplicate detection in the presence of retransmission. Numbering of octets within a segment is that the first data octet immediately following the header is the lowest numbered, and the following octets are numbered consecutively.

Typical kinds of sequence number comparisons which TCP must perform include determining:

(1) that an acknowledgment refers to some sequence number sent but not yet acknowledged.

(2) that all sequence numbers occupied by a segment have been acknowledged.

(3) that an incoming segment contains sequence numbers which are expected.

2.5. Establishing a Connection

```
     TCP A                                                TCP B

1.   CLOSED                                               LISTEN

2.   SYN-SENT    --> <SEQ=100><CTL=SYN>                --> SYN-RECEIVED

3.   ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

4.   ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>       --> ESTABLISHED

5.   ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED
```

Figure 5: Basic Three-Way Handshake for Connection Synchronization

Figure 5 illustrates the basic "three-way handshake" procedure used to establish a connection. This procedure normally is initiated by one TCP module and responded to by another. The procedure also works if two TCP modules simultaneously initiate the procedure. When simultaneous attempt occurs, each TCP module receives a "SYN" segment which carries no acknowledgment after it has sent a "SYN". The arrival of an old duplicate "SYN" segment can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of "reset" segments can disambiguate these cases.

2.6. Data Communication

Once the connection is established, data is communicated by the exchange of segments. Because segments may be lost due to errors or network congestion, TCP uses retransmission (after a timeout) to ensure delivery of every segment. Duplicate segments may arrive due to network or TCP retransmission. As discussed in the section on sequence numbers, TCP performs certain tests on the sequence and acknowledgment numbers in the segments to verify their acceptability.

The sender of data keeps track of the next sequence number to use in the variable SND.NXT. The receiver of data keeps track of the next sequence number to expect in the variable RCV.NXT. The sender of data keeps track of the oldest unacknowledged sequence number in the variable SND.UNA. If the data flow is momentarily idle and all data sent has been acknowledged, then the three variables will be equal.

When the sender creates a segment and transmits it, the sender advances SND.NXT. When the receiver accepts a segment, it advances RCV.NXT and sends an acknowledgment. When the data sender receives an acknowledgment, it advances SND.UNA. The extent to which the values

of these variables differ is a measure of the delay in the communication. The amount by which the variables are advanced is the length of the data in the segment. Once in the ESTABLISHED state, all segments must carry current acknowledgment information.

2.7. Closing a Connection

The TCP specification treats CLOSE in a simplex fashion. The user who CLOSEs may continue to RECEIVE until he is told that the other side has CLOSED also. Thus, a program could initiate several SENDs followed by a CLOSE, and then continue to RECEIVE until signaled that a RECEIVE failed because the other side has CLOSED. We assume that TCP will signal a user, even if no RECEIVEs are outstanding, that the other side has closed, so the user can terminate his side gracefully. A TCP module will reliably deliver all buffers SENT before the connection was CLOSED so a user who expects no data in return need only wait to hear the connection was CLOSED successfully to know that all his data was received at the destination TCP. Users must keep reading connections they close for sending until TCP says no more data will be received. The normal close sequence is shown in Figure 6.

```
    TCP A                                                    TCP B

1.  ESTABLISHED                                              ESTABLISHED

2.  (Close)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

3.  FIN-WAIT-2  <-- <SEQ=300><ACK=101><CTL=ACK>      <-- CLOSE-WAIT

4.                                                       (Close)
    TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

5.  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>      --> CLOSED

6.  (2 MSL)
    CLOSED
```

Figure 6: Normal Close Sequence

3. Implementation of TCP

MICROS is exploring ways to organize network computer to solve large problems. The overall goal is to develop a modular distributed operating system that can easily be ported onto a wide variety of network computers for development and execution of large and small distributed application systems[3][4].

The hardware for MICROS now consists of two LSI-11/23 nodes and seven MC68000 nodes, all linked by Ethernet channels. Figure 7 shows the configuration of MICROS.

A local operating system portion called SAM2S (Stand-Alone Modula-2) is resident in every node of the network. Modula-2 offers more extensive and efficient programming support environment[5]. SAM2S was developed initially to assess Modula-2 as a language for writing large systems and to provide portable software for Modula-2 programming support workstations.

SAM2S is a portable, highly modularized operating system. It offers extensive network communication facilities. A communication subsystem of 11 modules provides packet cast services and supports the DARPA Internet Protocol suit. SAM2S communication modules is shown in Figure 8.

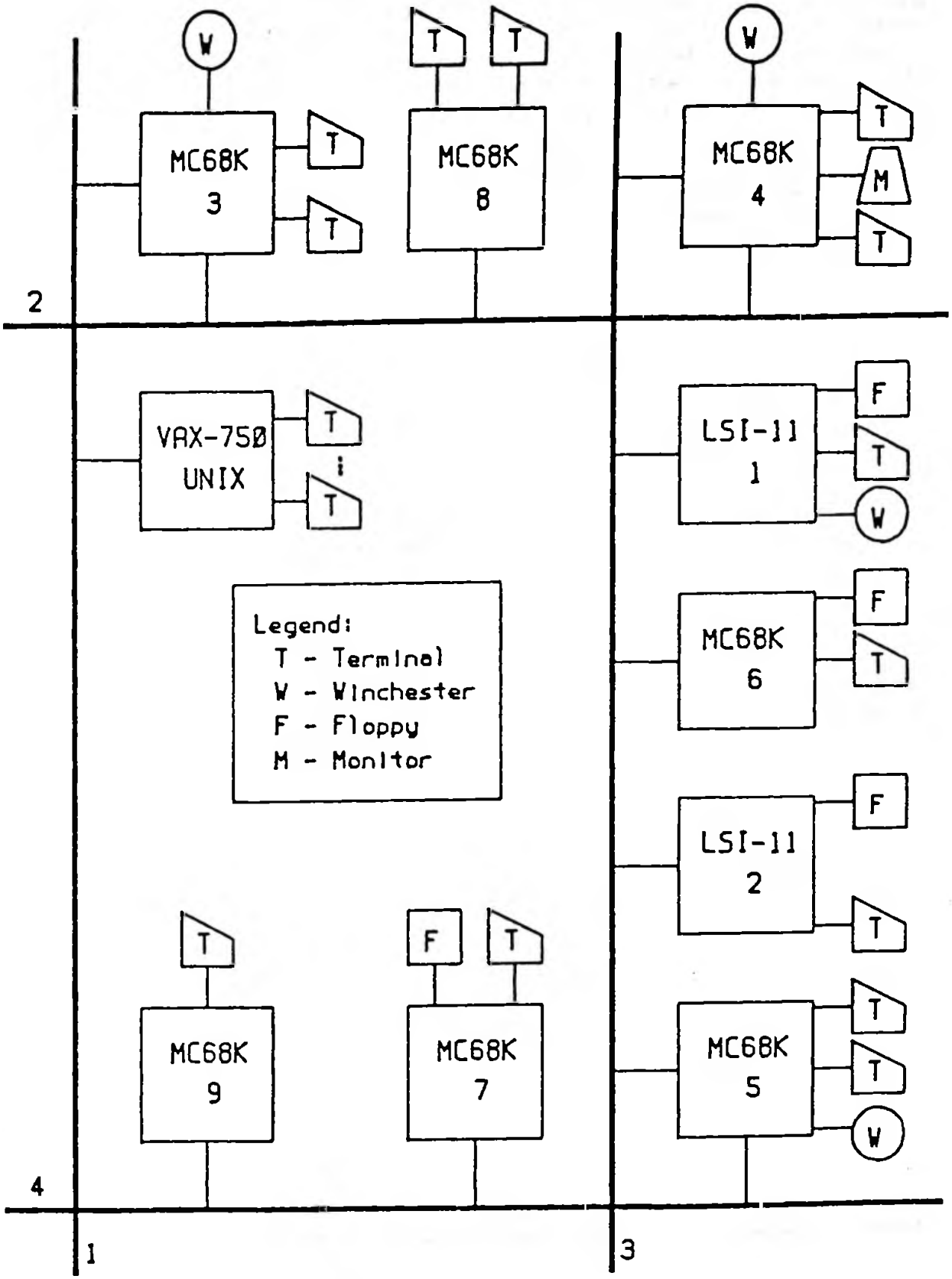


Figure 7: The MICROS Network

IP (Internet Protocol) unifies the available network services into a uniform internet datagram service. The IP includes such functions as a global addressing structure, provision for type of service requests, and provision for fragmentation of packets and reassembly at the destination host. IP packets are encapsulated and decapsulated by the IPheader module.

MARP (Multi-LAN Address Resolution Protocol) is used to convert a 32 bit IP address to a 48 bit Ethernet address. It works in an environment of multiple interconnected LANs.

UDP (User Datagram Protocol) provides basic datagram services and permits individual datagrams to be sent between hosts.

Tcp module in SAM2S communication subsystem contains mechanisms to provide reliable transmission of data. These mechanisms include sequence numbers, window management, acknowledgments, checksums, multiplexing, and retransmission procedures.

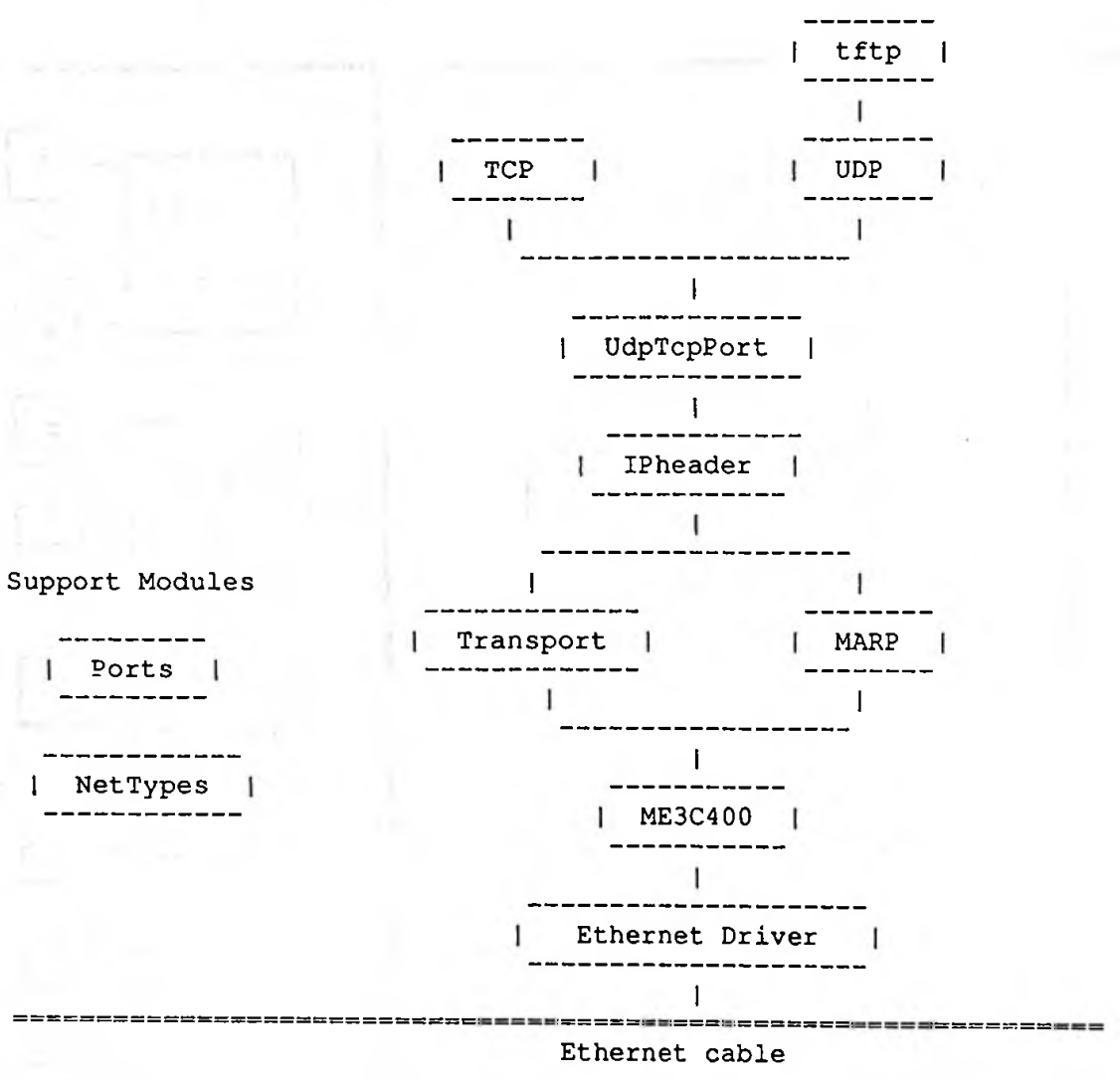


Figure 8: Communication Modules of SAM2S

All communication type definitions are present in the NetTypes module. ME3C400 is the device driver for the Ethernet controller. Transport is the main clearing house for packets as far

as user processes are concerned. Asynchronous exchange of packets is done by the Ports module.

The UdpTcpPort module implements the abstract entity through which all communication in the network takes place. It is inherently a bidirectional structure which is able to send and receive IP at the same UdpTcpPort. In some cases, server may listen to so-called well-known addresses, which are widely known and rarely changed. The UdpTcpPort is created and destroyed dynamically. The structure of UdpTcpPort is illustrated in Figure 9.

```
UdpTcpPort = POINTER TO UdpTcpPortRecordfield;
UdpTcpPortRecordfield = RECORD
                            ReceiveSnd, ReceiveRcv:  PORT;
                            PortNumber:  SHORTCARD;
                            Tcpcb:  ADDRESS;
                            Index:  CARDINAL;
                        END;
```

Figure 9: Structure of UdpTcpPort

When a connection is established, the command "Connect" creates a UdpTcpPort and Transmission Control Block (called Tcpcb in our implementation) that is linked to UdpTcpPort. After that, when a send on a UdpTcpPort is done the UdpTcpPort manager transfer the IP to the send port of Transport module. The Transport manager calls on the UdpTcpPort manager when it receives a IP, and places the IP on the receive port of the UdpTcpPort thus found.

The Tcp module is described in detail in the following sections.

3.1. Data Structure

We define Tcp record structure to correspond to the specification of the header format (see Figure 10). This structure includes the pseudo header.

```
TCP = POINTER TO TcpRecord;
TcpRecord = RECORD
                next, prev:  TCP
                src, dst:  InetAdr;
                x1, pr:  CHAR;
                len:  SHORTCARD;
                sport, dport:  SHORTCARD;
                seq, ack:  CARDINAL;
                OffsetFlags:  SHORTCARD;
                win, sum, urg:  SHORTCARD;
                TcpdataPtr:  ADDRESS;
            END;
```

Figure 10: Tcp Record Structure

Tcpcb is named TCB in the specification and is used to store the connection information. Each connection between two ports creates a Tcpcb. Figure 11 shows the Tcpcb structure and its meaning.

```
TCPCB = POINTER TO TcpcbRecord;
TcpcbRecord = RECORD
                  DestAdd:  InetAdr;    (* destination address *)
                  DestPort:  SHORTCARD;  (* number of destination port *)
                  State:  SHORTCARD;    (* state of this connection *)
                  SndUna, SndNxt, SndUp  (* send sequence variables *)
                  SndWl1, SndWl2, Iss:  CARDINAL;
                  SndWnd :  SHORTCARD;
                  RcvNxt, RcvUp,        (* receive sequence variables *)
```

```
    Irs :          CARDINAL;
    RcvWnd :    SHORTCARD;
    RcvAdv :    SHORTCARD;  (* advertised window *)
    SndMax :    CARDINAL;   (* highest sequence number sent *)
    Maxseq :    SHORTCARD;  (* maximum segment size *)
    Flags  :    SHORTCARD;  (* acknowledgment flag *)
    Rtt :    SHORTCARD;     (* round trip time *)
    Rtseq :    CARDINAL;    (* sequence number being timed *)
    Srtt :    CARDINAL;     (* smoothed round—trip time *)
    RcvTcpQ :    TCP;       (* receive data buffer *)
    TcpQ :    TCP;          (* send data queue *)
    RtranQ :    RTQU;       (* retransmission queue *)
    After :    TCPCB;       (* sequencing queue *)
END;
```

Figure 11: Tcpcb Record Structure

TranQ record (see Figure 12) is a retransmission packet pointer, which indicates when and how many times the retransmission packet retransmits. The item "tcphead" points to the packet.

```
RTQU = POINTER TO TranQRecord;
TranQRecord = RECORD
                timer:   CARDINAL;
                times:   SHORTCARD;
                tcphead:   TCP;
                next:     RTQU;
              END;
```

Figure 12: TranQ Record Structure

To speed up the process of searching the retransmission queue, there is a search index called the Rlist queue. From Rlist, the position of retransmission packet can be found in the retransmission queue. Figure 13 shows the Rlist structure.

```
RLIS = POINTER TO RlistRecord;
RlistRecord = RECORD
                DeAddress:    InetAdr;
                Soport, Deport:    CARDINAL;
                Time:    CARDINAL;
                next:    RLIST;
              END;
```

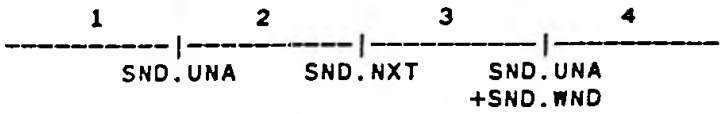
Figure 13: Rlist Record Structure

3.2. Window Management

The window mechanism is a flow control tool. Whenever appropriate, the recipient of data returns to the sender a number, which is (more or less) the size of the buffer which the receiver currently has available for additional data. This number of bytes, called the window, is the maximum which the sender is permitted to transmit until the receiver returns some additional window.

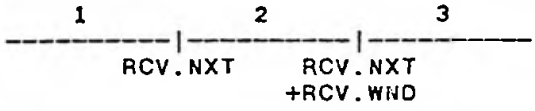
The send window is the portion of the sequence space labeled 3 in Figure 14. The receive window is the portion of the sequence space labeled 2 in Figure 15.

We define the window size is 2048 bytes in this implementation of TCP.



1. old sequence numbers which have been acknowledged
2. sequence numbers of unacknowledged data
3. sequence numbers allowed for new data transmission
4. future sequence numbers which are not yet allowed

Figure 14: Send Sequence Space



1. old sequence numbers which have been acknowledged
2. sequence numbers allowed for new reception
3. future sequence numbers which are not yet allowed

Figure 15: Receive Sequence Space

3.3. Arriving Segment Process

Two procedures: InputTcp and ProcessOther, which are called by Transport module, process the arriving segment. If the state of connection is CLOSED, LISTEN, and SYN-SENT, the InputTcp procedure is called. For other states of connection, the Transport module calls the ProcessOther procedure.

InputTcp and ProcessOther provide such operations as connection synchronization, examination of sequence number, window management, receiving data, acknowledgment and processing control flag. After data arrive, the ProcessOther procedure puts the data into the receive data buffer and acknowledges the data packet.

3.4. Retransmission Process

Because TCP provides the ability to transmit and receive data reliably, retransmission is one of most important functions of the Tcp module. After each data packet is sent, this data packet is linked to the TranQ table and the TranQ record is inserted into the retransmission queue. At the same time, a Rlist record is put into the Rlist queue. If the Tcp module gets a positive acknowledgment from the receiving segment, the procedure RemoveRtqu deletes the acknowledged packet and Rlist entry from the queue.

When the Tcp module is initiated, the Tcp module starts a process called RetransmitProcess. RetransmitProcess checks the timer of TranQ table once every logical time unit. When the timer gets the value which is defined by retransmission timeout, the RetransmitProcess finds the place of the timeout packet from the Rlist queue and transmits the data packet again. The Times module provides the logical time for RetransmitProcess.

3.5. User Commands

The Tcp module provides five user commands for user: Connect; SendTo; ReceiveFrom; Close and Status. These commands have been tested by the SeTest and ReTest modules.

3.5.1. Connect

Format:

Connect (Sport, Dport: CARDINAL; Dadr: InetAdr; OpenType: ConnectType);

Sport:	number of source port. If source port is not a well-known port, Sport is zero.
Dport:	number of destination port. If OpenType is Passive, Dport is zero.
Dadr:	destination internet address.
OpenType:	connection type, Active or Passive.

The Connect command creates a UdpTcpPort record and Tcpcb block. If OpenType is set to Passive, then this is a call to LISTEN for an incoming connection. A passive connection can be made an active connection by the subsequent execution of a SendTo. On an active connect command, the Tcp module will begin the procedure to synchronize (i.e. establish) the connection at once. If the connection is not created within the timeout period, the Connect command will return the signal "connect time out".

3.5.2. SendTo

Format:

SendTo (Daddr: InetAdr; Sportnu, Dportnu, Length: CARDINAL; Daptr: WORD);

Daddr:	destination internet address.
Sportnu:	number of source port.
Dportnu:	number of destination port.
Length:	data length.
Daptr:	pointer to the user data buffer.

This call causes the data contained in the user data buffer to be sent on the connection. If the connection has not been opened, the SendTo is considered an error.

3.5.3. ReceiveFrom

Format:

ReceiveFrom (Aaddr: InetAdr; Sportn, Dportn: CARDINAL);

Aaddr:	destination internet address.
Sportn:	number of source port.
Dportn:	number of destination port.

The ReceiveFrom command gets data from a receive data buffer and sends it to the user. In our implementation, this command displays the data on terminal.

3.5.4. Close

Format:

Close (Add: InetAdr; Spo, Dpo: CARDINAL);

Add:	destination internet address.
Spo:	number of source port.
Dpo:	number of destination port.

This command causes the connection specified to be closed. Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time.

3.5.5. Status

Format:

Status (Ad: InetAdr; Ps, Pd: CARDINAL);

Ad: destination internet address.

Ps: number of source port.

Pd: number of destination port.

After the Status command is executed, state information is returned from the Tcpcb associated with the connection.

3.6. Current Implementation of SAM2S with TCP

We have implemented most parts of TCP except dynamic timeout, precedence and security. Because of the variability of the networks that compose an internetwork system and the wide range of uses of TCP connections, the retransmission timeout must be dynamically determined. The MICROS communication system is a local network, so the Tcp module does not have to support dynamic timeout values for retransmission of packets in a byte-stream. Security is also not an important issue in a research environment.

3.7. Test Results

To test the Tcp module, SeTest and ReTest modules were used. Both of them can send and receive packets in workstations, but they have different numbers of ports. A user can give sample TCP commands on the workstation keyboard.

First, the command "Connect" is used, which establishes a connection. Then the SeTest module uses "SendTo" to send packets. The station in which the packet is received runs "ReceiveFrom" command to display the content of packets. After the communication is finished, the "Close" command is used to close the connection. We sent 30 packets in different sizes for the Tcp or IP module operation between two stations. Figure 16 shows a comparison of transmission speed for the Tcp and IP module.

	data length (Bytes/packet)	total time (sec)	speed (Bytes/sec)
TCP	128	36.1	106.37
	512	52.22	294.14
	1024	75.05	409.33
IP	128	15.33	250.49
	512	20.97	732.47
	1024	29.18	1052.77

TCP:		
	fixed cost/packet	0.982 - 1.03 sec/packet
	extra variable cost/KBytes	1.39 - 1.48 sec/KBytes
IP:		
	fixed cost/packet	0.43 - 0.48 sec/packet
	extra variable cost/KBytes	0.48 - 0.53 sec/KBytes

Figure 16: Test Results

Considering these results, it appears the transmission speed of the Tcp module is slower than that of the IP module because the Tcp module has additional processing related to its acknowledgments and window updates which consume much time. Thus the system overhead of the TCP module is higher.

Acknowledgments

The authors are grateful to Benoy de Souza for providing the IP implementation on which TCP is based. We would like to thank Weimin Zheng who provided the UDP and UdpTcpPort modules. We owe special thanks to Susan Choudhari for her excellent assistance in editing this paper, plus her many other contributions.

This work has been supported in part by National Science Foundation research grant MCS84-01624, CER grant DCS83-19966 and equipment grant MCS82-03955, National Aeronautics and Space Administration grant NAG-1-249, Army Research Office contract DAAG-29-82-K-0103 and instrumentation grant DAAG-29-84-G-0011, and an external research grant from Digital Equipment Corporation.

References

[1] B.M. Leiner, R. Cole, J. Postel, D. Mills, "The DARPA Internet Protocol Suite", IEEE Communications Magazine, Vol.23, No.3, March 1985.

[2] J. Postel, "Transmission Control Protocol", RFC-793, USC/Information Sciences Institute, Sep. 1981.

[3] L. D. Wittie and A. J. Frank, "A Portable Modula-2 Operating System: SAM2S", Proc. AFIPS National Computer Conference, July 1984.

[4] Larry D. Wittie, "Distributed Operating System Methods for Large Network Computers", Proc. 1st Pacific Computer Communication Symposium, Oct. 1985.

[5] N. Wirth, *Programming in Modula-2,* Springer-Verlag, 1st edition 1982, 2nd edition 1983, 3rd edition 1985.

Building an Operating System with Modula-2

Brad Justice
Stan Osborne
Vivian Wills

Computer Science Department
San Francisco State University

ABSTRACT

An operating system kernel for the IBM PC and an interactive application written in Modula-2 are presented. The functionality provided by the operating system kernel supports multi-programming with a preemptive scheduler, device interface, queue management, and a timeout request server. The kernel is complete enough to develop interactive, real-time, and concurrent processing applications. It allows a design method based on co-operating Processes and Monitors to be used for building useful applications. The kernel and the application design technique are presented.

An application using the kernel is also discussed. The application allows multiple data files to be copied simultaneously using multi-processing. A part of this application is a window subsystem. The window subsystem allows the input and output from a menu process to appear at the top of the display. A window in the lower part of the display is used for output data.

May 24, 1986

CONTENTS

Building an Operating System with Modula-2

Brad Justice
Stan Osborne
Vivian Wills

Computer Science Department
San Francisco State University

1. Introduction

Operating Systems have long been considered one of the most difficult areas of computer programming. In the last few years new analytical methods and software development tools have been developed that make building operating systems much easier. Some of these new tools and methods were used by one section of the "Operating System Principles" classes taught at San Francisco State University in the Spring Semester of 1985. The main goal of this class was to provide the students with an understanding of the components of operating systems. This article is a description of the the design method taught to the students, the operating system kernel used in the class project, and an application system developed using the kernel.

Limited hardware was available for the class to use for hands-on system development. The students have access to IBM PC's in the Computer Science Department's Laboratory, other public computing labs on campus, personal systems at home, or systems at work. This provided enough access for all class members to use standard IBM PC hardware with two floppy disk drives for a class project.

The Modula-2 programming language was chosen for its features that allow concurrent programming, interface to device interrupts, and management of software modules developed by a group of programmers.

The students were provided with a design method for the development of interactive application systems. The method allows programming applications to be divided into their logical components so they can be analyzed for correctness and efficiency before they are built. When a design for an interactive system is complete the components of the system to be built are clearly defined and each can be easily built.

1.1 Terminology

One confusing aspect of communication among system programmers and designers today, is a lack of a common way for them to describe the work they do and the tools they use. Many of the words used in this paper have specific meaning when used to describe computer based operating systems and their applications. To help understand the words we have used, a glossary is provided at the end of the article. Many of the definitions found in this glossary are based on definitions adopted by the International Standards Organization (ISO) to standardize computer terminology.

1.2 Project Goal

The goal of the class project was to provide the students with practical experience in the design of an efficient and maintainable system programming application. The goal was to provide the students some experience with the problems of building operating systems and concurrent applications. To achieve this, a project was devised that required the use of interaction with people, the interface of software to existing hardware devices in a non-standard way, and the use of a pre-emptive, multi-processing, operating system kernel. In addition to this the students were required to work in groups of two or three, each group building their own version of the class project. To make this project easier to achieve the project components were defined in a way that allowed them to be divided into three parts. Each member of the group worked on their part of the total effort.

The application assigned was a file transfer utility. This utility had to allow multiple files to be copied simultaneously to different devices while being controlled from a single menu. This required the interaction with people to occur concurrently with the transfer of files.

The specifications for the application also required that a menu of options be displayed in one part of the screen while data from a file was displayed in another part of the screen. This requirement meant two or more processes had to do I/O to the screen at the same time with each process having its own display window.

The students were taught how to analyze the project requirements and to divide these requirements into separate components. The types of components needed were generalized to "Processes" and "Monitors". Each project group was provided with an O/S kernel written in Modula-2 to use as a part of the assignment.

1.3 The IBM PC

The system kernel and file transfer application were developed and run on the IBM PC. Program development and testing was done using readily available software tools and a standard hardware configuration.

1.3.1 Hardware Standard IBM PC hardware was used to develop the operating system and application. The minimum hardware configuration was an Intel 8088 Microprocessor with at least 256 KB RAM, two 360 KB floppy disk drives, a monochrome controller and monitor.

1.3.2 Software Standard IBM PC software was used to develop the operating system. IBM PC ROM BIOS, IBM PC-DOS V2.0. with the statement DEVICE=ANSI.SYS included in CONFIG.SYS, The Modula-2 compiler from Logitech for the IBM PC was used for all software development. The early Modula-2 development was done with Release 1.10 from Logitech. Recently all modules have been compiled and tested using Release 2.00.

IBM PC-DOS and ROM BIOS routines are used both directly and indirectly by the application. All file operations (open/close, read/write, etc.) and the reading of the current time of day were done using Modula-2 library routines supplied with the Logitech compiler. Whenever possible the application has used Modula-2 procedures rather than direct calls to DOS or ROM BIOS. In a few cases specific features or better performance were needed. For these cases the application used the inline assembly language features of the Modula-2 compiler to call DOS or ROM BIOS directly. Routines that needed to use assembly language to call ROM BIOS were the printer output routines and the windowed video output.

1.3.3 Alternate Configurations - Hardware/Software The system kernel, the application, and the Modula-2 development tools have also been tested on the following systems: AT&T PC 6300, IBM PC-AT, ITT XTRA XP, Kangaroo AT. Logitech Modula-2/86 release 2.00 for the IBM PC runs without trouble on these systems.

To put this software on a different hardware system would require changing those parts that are dependent on the BIOS, DOS, or written in assembly language. The operating system kernel has no instructions that are hardware or operating system dependent. The device driver and the application modules are the modules that contain hardware and system dependent routines.

2. *Building Systems*

A good system can only be built from a good design.

A good design is developed by repeating design development steps. These steps consist of: analysis of the requirements, a survey and analysis of the available tools, a top down design, review, and analysis. With each repetition, the design becomes more refined. The refinement continues until the design is complete enought for construction to begin. The decision to begin construction is done only after the important system components are clearly described by the design. After construction has started, the design may be further improved and refined. The continued improvement must be done within clearly identified constraints.

The repeated analysis, redesign, decomposition, and refinement of the design allows systems to be divided into components based on a few simple programming structures. When the design is completed and a description for each component exists, the programming structure used in building the component is also known.

Once the final system has been described as simple, understandable, and easy to build components, the programming and testing of the system begins.

The design components described here are based on those proposed by C.A.R. Hoare in 1973. Systems are decomposed into three basic components: "Process", "Class" and "Monitor". Each of the three basic component types has its own definition and a specific way to be programmed. By dividing a system into these components many benefits are achieved. The most important benefit is that analysis can be done on the individual components and each component can be built and tested independently of the final system.

2.1 The Functional Specification

When a new system is to be built a description of the work to be done by the system is needed. A "Functional Specification" is a written description of what the system is to do when completed. The functional specification is used for testing and evaluating the development. It is the only record of what the developers are trying to build.

A good functional specification will say little about how the system is to be developed. Instead the specification describes what a user of the system will expect as output from the new system. A functional specification is often a short document since the output needed by a user is often limited.

The specification must be clearly written so the person testing the new system can decide if all the required functionality has been completely developed and works correctly.

2.2 The Analysis

This is the process of dividing the desired system into components that can be clearly described and built. The needed features of the new system are used to determine the components of the system.

First an initial design is conceived. Two types of analysis are used; the flow of data and the flow of processing control. From successive iterations of analysis and design, a final design is derived. If an error is detected during testing of the design, further analysis will provide a way to change the design to correct for the error.

2.3 The Design

The highest level of description for the system design structure is a drawing. This drawing describes the flow of processing control and is drawn with circles and arrows. A circle describes a programming step. An arrow describes the transfer of program control from one step to another. (A program step is a group of instructions that do useful work by controlling a machine.)

If data is transferred between steps, the transfer of data occurs at the time control is transferred. The actual transfer of control between one programming step and another can be used to convey information to the second programming step. A designer can use this implied information to prevent the unnecessary transfer of data between processing steps. Usually the transfer of control is programmed as a call to a procedure and the data transfer is done with parameters to the procedure. (With this method of design for program structure the the flow of data is not explicitly shown.)

The drawings are used to show how the various pieces of the system connect with each other. Analysis can be done with the information contained in the picture without programming the application. The main result of this method is an easy to build design. This method of describing a design allows a system to be designed that runs with reasonable and predictable performance characteristics.

Design Diagram Elements

Circles are used to represent the program components of the system. Arrows are used to represent the transfer of processing control from one program component to another. Each component has its own local data and an initialization routine.

The Modula-2 programming language is well suited for programming these components. The language facility for creating a "DEFINITION" module for a system component that is separate from the implementation allows the interface between components to be specified before they are programmed. Also the existence of a "DEFINITION" module for each of the shared components guarantees the interface will be the same for all references to the shared component.

Note: Other properties of this design technique are esoteric in nature and of interest to Computer Scientists. The design method allows the components to be put into a hierarchical order. This ordering results in a system that can be studied as a sequence of abstract machines simulated by programs. Mathematical induction can be used to prove general properties of the system (such as the absence of deadlocks). The efficient use of each resource is possible by ordering components according to the speed of the physical resources they control. Stochastic models can be developed to model the systems dynamic behavior. ([DIJ71],[BRI73])

2.3.1 Process A sequential set of programmed instructions for a machine that achieve a well defined result. A process may transfer control to other components, but the other components may not directly transfer control to a process.

A process is drawn as a circle with zero or more arrows pointing out of the circle.

A Process Diagram.

2.3.2 Class A class is a program component that is used repeatedly and perhaps duplicated each time it is needed. Control is transferred to a class from other program components. A class may pass control to other program components. The most common form of a class is as a subroutine library. Each time a component needs the functions of a class, the data that define the class are duplicated.

With the Modula-2 language it is possible to build classes that are reentrant. This allows a single copy of the class instructions to be shared among all the components that need the class.

A class is drawn as a single arrow leading to a circle with zero or more arrows pointeding out of the circle.

A Class Diagram.

2.3.3 Monitor A monitor is a program component that is shared by other program components in a system. A monitor shares a resource between competing program components. Control is transferred to a monitor by a program component when it needs to use the resources controlled by the monitor. A common use for a monitor is to share a memory buffer between an interrupt process and an application process.

A monitor is drawn as two or more arrows leading to a circle with zero or more arrows pointing out of the circle.

A Monitor Diagram.

A monitor is shared between two or more program components and it must prevent access to data by one component from interfering with access to the same data by another component. This requirement is called "mutual exclusion". When this requirement is met only one component may have access to a shared resource at a time. If a second component wants to access the shared resource while the first has access, the second component must wait for the first to finish using the resource. When a monitor controls program mutual exclusion, the enforcing of mutual exclusion is contained in a single set of instructions. A monitor allows mutual exclusion to be localized to single component instead of being distributed among all the components sharing the resource.

2.4 Other Tools

Certain system building tools are needed in almost every application. These are often defined or built in a general way and become tools for use in building applications.

System builders often have available to them procedures that provide frequently used services. Often a general queue management service that allows for the use of queues, a system kernel for the sharing of a single CPU among many processes (programming steps), and for the synchronization of co-operating processes are provided to the builder of software controlled systems.

2.5 Component Grouping

The different types of system components are often grouped together to organize their analysis and description. One way of grouping components is called "layering". At the lowest level are the hardware devices, followed by the device interface software, the system services software, the system interface classes, and finally the application processes.

Another way of grouping components is by dividing the components into sub-systems. This method of grouping uses a close relationship between the components to group the components. The name given to the sub-system usually describes the relationship between the components. Sub-systems grouped this way often have names like: "The File System", "The Network Interface", "The User Interface".

2.6 Design Refinement

This is an activity that goes on during the entire time a system is being built. The designers of the system are constantly looking for ways to improve their design. A system designer uses analysis, design methods, knowledge about the requirements for the new system, and any other available information to develop an initial design. Once an initial design exists in any form, the refinement of the design begins.

Through a cycle of analysis, refinement, and re-analysis the design is steadily improved. From the first design more analysis can be done. This leads to a better understanding by the designer of the system being built. The improved understanding of the design and the system to be built give the designer more insight about how to improve the design. Each time the design is improved the design that results is a refinement of previous designs.

Design Refinement occurs in four phases during the building of the system described by the design. In each phase different types of refinement are done. The refinement process starts with coarse changes in the design and ends with fine improvement. Throughout the building of the new system, the design is improved. These phases of refinement are: before any construction, during the construction of the structure, during filling in the structure, and during the final testing of the system.

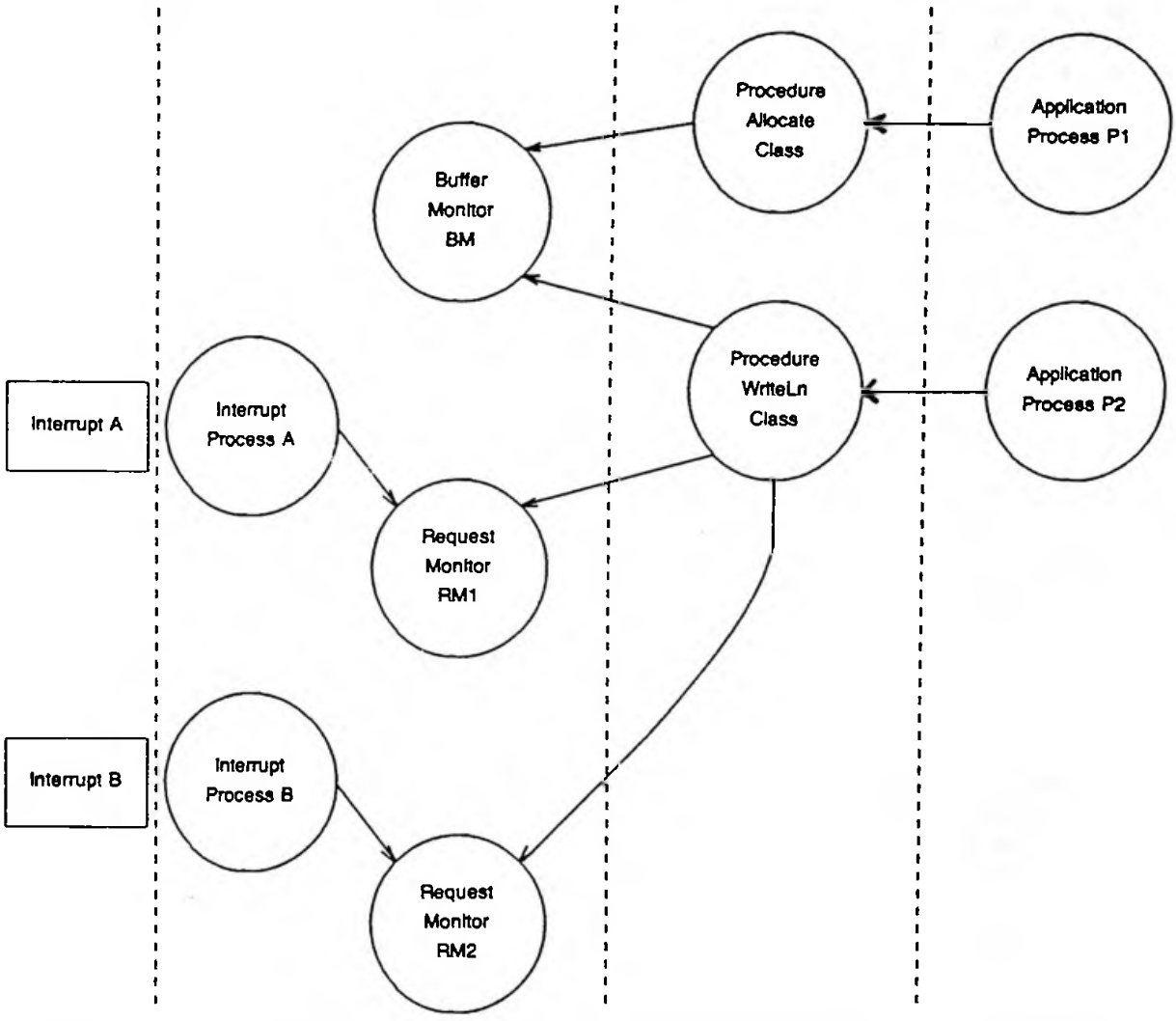
During the first phase of refinement the major processing functions of the system are identified and the main data inputs and outputs are described. This includes describing the main processes, classes, and monitors needed to buid the system. These components are the general structure of the new system. Also some limitations in the hardware functionality can be compensated for by the addition of monitors that supply the missing functionality. This is the best phase for making major changes in the structure of the design.

The second phase is the interface between the components of the design and the internal structure of the system are defined and built. From this point on when a design problem is identified the analysis of the problem must include the difficulty (cost) of redoing all or some of the work completed so far. If the design created in the first phase is a good design, any problems discovered during the second phase can be corrected by improving the existing design.

The third phase is the detailed programming effort takes place. All functions of the new system are programmed. During this phase its difficult to justify making a major change in the design. Any changes made to the structure of the design at this time should be minor.

The fourth phase is the new system is heavily tested. Only if serious defects in the design are detected, will any change be made to the design. The ideal for this phase would be that no defects in the design are uncovered by the testing process. Only defects in how the system was built should be discovered at this time.

2.7 An Example System Design



2.8 Building the Design

This is the activity of converting the design (one or more diagrams and descriptions of the processing steps) to a running program. Each type of processing step is built in a specific way using Modula-2.

First the "DEFINITION" modules need to be developed for all the class and monitor components of the target system. These components are easy to identify as they are the

ones with arrows leading into them. Once these "DEFINITION" modules are completed these components should be typed in and tested.

Then the system initialization sequence is programmed. This is the part of the system that makes sure all the processes are started in the right sequence. Once the instructions for initialization of the system are completed only the programming of the individual processes remains.

With the Modula-2 programming language there is a slight difference between the programming of a process and an interrupt process. A normal process loops according to the way it was programmed. An interrupt process executes one iteration of a processing loop for each interrupt from the hardware device.

3. *The Operating System Kernel*

The operating system kernel is a software tool consisting of Modula-2 procedures, types, variables and processes that schedule the execution of concurrent processes. Concurrent processes are programs designed to be executed simultaneously. They may be executed simultaneously on two or more processors. This is true concurrency. Or more commonly they are executed on a single processor that switches between the different processes. This is called quasi-concurrency.

The IBM PC has a single processor so true concurrency does not happen. The processes execute quasi-concurrently. One process is given the processor and some of the instructions of the process are executed. The process may become blocked. This means it encounters some condition that makes it impossible for it to continue doing useful work. If so, the current process relinquishes the processor to another process. If the process is never blocked, it will continue to execute until its time limit expires. When this occurs, a hardware clock is used to interrupt the process and and start an interrupt process in the kernel. The interrupt process then gives the processor to a second process. This pattern repeats, the second process then executes until it too relinquishes the processor or is timed out. Eventually the first process is again given the processor and it resumes from where it was interrupted.

The alternative to this approach is sequential programming. In sequential programming there is a single process. The instructions of the single process are executed in a continuous sequence. The processing of the program can not easily divide its I/O operations into separate processes that overlap with each other or with computational tasks.

A Kernel allows the IBM PC to do what is done on larger time-sharing systems. On such a system there are multiple users or processes at a single time. The processor is switched back and forth between each process so rapidly that it appears as though there is a processor dedicated to each process.

There are many advantages to concurrent programming instead of sequential programming. Multiple tasks can run simultaneously when concurrent programming is used. For example, the application presented in this article can interact with the user while it is transferring files.

Another advantage is that the system is made more efficient by avoiding "busy waiting" programming techniques. Sometimes a process must wait for a particular event, often this is for information to be entered by the user. In a sequential program this would usually be programmed with a software loop. The program repeatedly tests to see if the event has occurred. In a concurrent program the waiting process relinquishes the processor, allowing another process to execute and do useful work. When the awaited event occurs the waiting process resumes execution. With concurrent programs when one process is waiting the others can continue their execution.

In addition to sharing the processor among many processes the kernel provides the mechanism for controlling access to other shared resources. These resources may be hardware resources, such as a printer, or software resources, such as service routines or shared data.

The scheduling of concurrent processes can be conceptualized as the allocation of shared resources. The most important of these resources is the processor that executes the processes. The kernel provides the mechanism that allocates the processor to a process so the instructions of the process are executed. The IBM PC has one processor and there may be many concurrent processes ready to begin execution. The processor resource must therefore be shared by the processes.

3.1 The Kernel

Our operating system kernel is written in Modula-2. The Modula-2 language provides many higher level features useful in writing a system kernel. Several of these deserve mention at this point.

PROC A data type. A PROC is a parameterless procedure.

PROCESS A standard type for a process. In Modula-2 a PROCESS is a coroutine for use with a kernel that supports many processes sharing one or more processors.

Note: This definition corresponds to the official definition for a PROCESS at the time the Kernel was built. This definition has been altered in the most recent edition (third) of *Programming in Modula-2*.

NEWPROCESS `NEWPROCESS(P:PROC;A:ADDRESS;n:CARDINAL;VAR new:PROCESS);`

A standard procedure to create a process. NEWPROCESS is passed P, the address of the instructions to be executed; A, the location for the PROCESS workspace; and, n, the size of the workspace for the PROCESS. NEWPROCESS returns new, the process created.

ALLOCATE `ALLOCATE(VAR a:ADDRESS; size:CARDINAL);`

A standard process that allocates memory space. ALLOCATE sets aside memory. For example, a workspace is needed when NEWPROCESS is called. ALLOCATE is passed the size of the area to be allocated and returns a, the location of the area allocated.

DEALLOCATE `DEALLOCATE(VAR a:ADDRESS; size:CARDINAL);`

The opposite of allocate, DEALLOCATE is passed the address and size of an area in memory to be made available for reuse.

TRANSFER `TRANSFER(VAR p1,p2:PROCESS);`

A standard procedure, TRANSFER passes control of the processor to a PROCESS. There are two parameters: a source and a destination process. When source PROCESS p1 calls TRANSFER its execution is suspended and the execution of destination PROCESS p2 is resumed at the point where it was last interrupted. This procedure is used pass control of the CPU from one process to another.

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

Similar to TRANSFER with one important difference, this routine receives control of the CPU from a hardware interrupt. To do this there is a third parameter, va, the interrupt vector value. On execution IOTRANSFER passes control between a source and a destination PROCESS; in addition the source PROCESS p1 is installed as an interrupt handler at the vector va. The occurrence of the interrupt will cause the resumption of the source PROCESS (the interrupted process) at the instruction immediately following IOTRANSFER.

The kernel uses these Modula-2 facilities along with standard pointer operations for the creation, scheduling and destruction of concurrent processes.

For the kernel a new concept of a process is required, different from the Modula-2 PROCESS. This process consists of a PROCESS plus a process descriptor. The process descriptor is a RECORD that stores information for the scheduling and eventual destruction of the process.

The definition of the process descriptor is as follows:

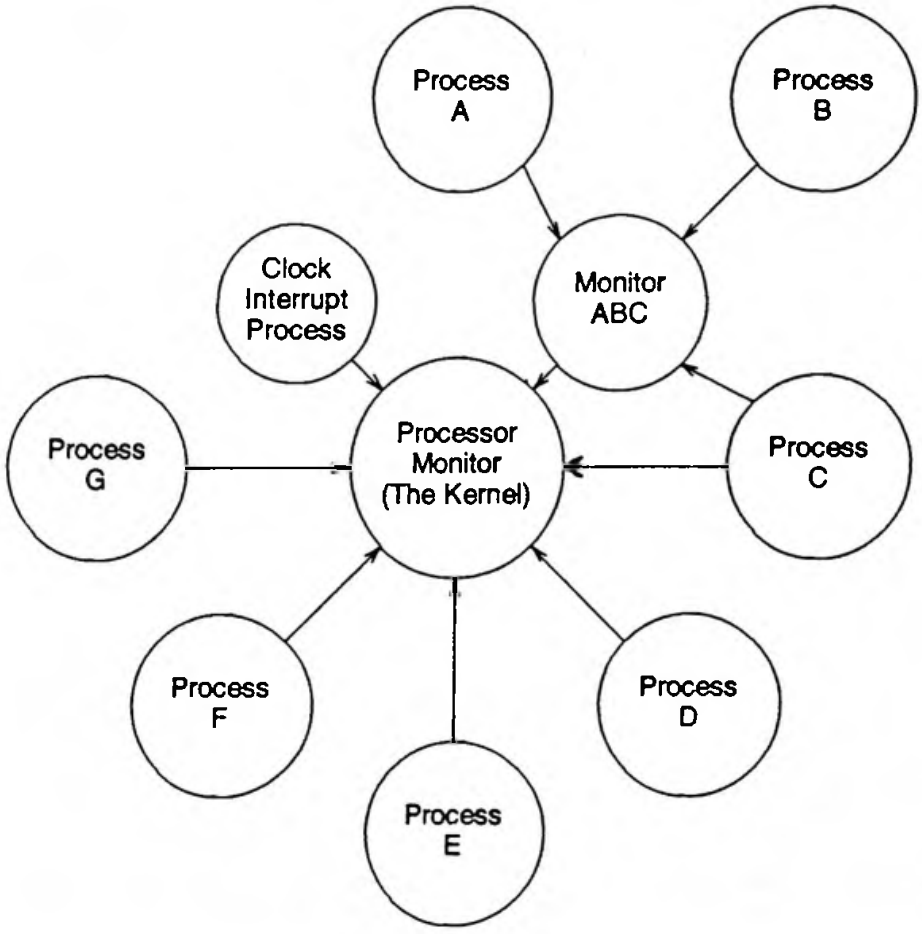
```
TYPE Processdescriptor =
                RECORD
                  Next: Pdpointer;
                  Cor: PROCESS;
                  Corsize: CARDINAL;
                  Sleepcount: CARDINAL
                END;
```

where Pdpointer is a POINTER to a Processdescriptor.

"Cor" can be thought of as a pointer to the PROCESS to which the Processdescriptor applies. "Next" can be used to build linked lists of Processdescriptors.

The Cp is a Pdpointer. It marks a special process, the current process. This is the process that has been allocated the processor and is executing the instructions of its program.

The Kernel and its relationship to the other components of a system can be drawn as follows:



3.1.1 Queue Management An Eventqueue is a linked list of Processdescriptors that are waiting for some event. Its definition is:

```
TYPE Eventqueue = ARRAY [Top..Bottom] OF Pdpointer;
```

Two Pdpointers are used; one points to the top of the queue, the point from where processes are generally removed from the queue, and one points to the bottom of the queue, the point where processes are usually added.

There are two Eventqueues used by the kernel. These are the Ready queue and the Sleepqueue. The Ready queue consists of processes waiting for the processor. The Sleepqueue consists of processes that have elected to remain inactive for a certain period of time. They are waiting to be awakened by the kernel.

In addition to these two Eventqueues there can be many others. Eventqueues can be created as required by an application program; there are many such Eventqueues in the application described in this paper.

There are four operations that can be done on TYPE Eventqueue.

Awaited `Awaited(S:Eventqueue):BOOLEAN;`

Awaited is a function used by a process to test an Eventqueue for a Waiting process. If Eventqueue S contains a process Awaited returns TRUE. If the Eventqueue is empty it returns FALSE.

Init `Init(VAR S:Eventqueue);`

Initialization of the Eventqueue. This must be done to an Eventqueue before any other operation.

Signal `Signal(VAR S:Eventqueue);`

When a process calls Signal it starts the top process in the Eventqueue S. If Eventqueue S contains no processes, Signal is a null operation. Otherwise the top process is removed from the Eventqueue and placed in at the bottom of the Ready queue. The next process in the Eventqueue, if any, becomes the top process.

Wait `Wait(VAR S:Eventqueue);`

The process calling Wait relinquishes the processor and enters Eventqueue S. If there are processes already waiting it is added to the bottom of the queue. It does not become an active process until it rises to the top of the Eventqueue and is Signalled by some other process.

3.1.2 Processes A process is created through a call to Startprocess. There are two parameters: Size, a CARDINAL, and Routine, a PROC. Size is the amount of memory that must be allocated for the execution of the new process and Routine is the address of the instructions that will be executed. Through calls to ALLOCATE memory locations for the Processdescriptor and the PROCESS are allocated; the PROCESS is created through a call to NEWPROCESS; finally the new process is placed at the bottom Ready queue for eventual execution.

The need to specify the workspace size is one difficulty of using Modula-2 for co-processing. It can be difficult to assess in advance the space requirement of a process. Specifying a space too large wastes memory while specifying a space too small can cause the system to crash.

The new process waits in the Ready queue while those processes ahead of it are removed from the queue and executed. Eventually it rises to the top of the queue. It is removed from the queue and becomes the Cp, the current process. It is allocated the

processor and the instructions of Routine are executed. It continues to execute until it is timed out or it calls one of the four PROCEDURES that result in it relinquishing the processor to the next Ready process. These four PROCEDURES are:

Finishprocess `Finishprocess;`

The calling process is destroyed, its memory space made available for reuse through calls to DEALLOCATE , and the process at the top of the Ready queue becomes the current process. "Corsize" is stored in the Processdescriptor so Finishprocess can pass it as a parameter to DEALLOCATE. After the calls to DEALLOCATE, TRANSFER is called to allocate the processor to the new current process.

Pause `Pause;`

The calling process is placed at the bottom of the Ready queue and the process at the top of the Ready queue becomes the the Cp. A call to TRANSFER allocates the processor to the new current process.

Sleep `Sleep(Count:CARDINAL);`

The calling process is placed in the Sleepqueue where it remains for Count clock pulses. Again, the process at the head of the Ready becomes the Cp and the processor is allocated to this new current process by calling TRANSFER. After Count clock pulses the calling process is removed from the Sleepqueue and placed at the bottom of the the Ready queue by the kernel.

Wait `Wait(VAR S:Eventqueue);`

This calling process is placed at the bottom of Eventqueue S. The process at the top of the Ready queue becomes the Cp. A call to TRANSFER allocates the processor to the new current process.

If a process does not call any of these four PROCEDURES it will continue to execute until its allotted time expires. The kernel places the process at the bottom of the Ready queue. The process at the top of the Ready queue becomes the Cp and the process is allocated the processor through a call to IOTRANSFER. The ability to interrupt the current process and to allocate the processor to a new current process is called preemption. The preemptive mechanism of this kernel is discussed under "Device Interrupts" below.

If a process is timed out and there are no processes in the Ready queue the process is given another time period and continues to execute. This is also what occurs if Pause is called. If Wait, Finishprocess or Sleep are called when there are no processes in the Ready queue the system will idle until some process enters the queue. That process then is made the Cp and is executed.

A new process enters the Ready queue when Startprocess is called. From this point until its destruction the Processdescriptor will be in one of three locations. Each location corresponds to a process state. First, the process could be the Cp, the current process. If so, its state is "running"; its instructions are being executed. Second, a process could be in the Ready queue. When its status is "ready" it could execute if it were allocated the processor. Finally, a process could be in some other Eventqueue where its state is "blocked". This prevents it from continuing until an event occurs.

3.1.3 Device Interrupts There is another class of processes that a kernel must contend with: unscheduled processes. These processes are not executed because they go to the head of the Ready queue. Instead they are executed in response to an external unscheduled event caused by hardware. These processes are usually called interrupt handlers. When an interrupt occurs, execution of the current process is suspended and the interrupt handler

process is executed.

Interrupt handlers are not processes like those previously discussed. They have no process descriptor. They wait in no Eventqueue for processing. They can not be timed out. On completion of their task they do not pass control through a call to TRANSFER, they use IOTRANSFER instead.

Note: Following the recommendations of Logitech an effort was made to keep to a minimum the instructions executed by the interrupt processes. This makes particular sense with the clock handler, which is executed 18 times per second. Sometimes the interrupt handler does tasks that could be done through a call to kernel procedures. Instead the instructions are included in-line. This is done to avoid the overhead inherent in a procedure call.

3.2 Clock Device Interface

One interrupt handler is required for the execution of the operating system kernel: the clock interrupt handler. Other interrupt handlers can be created as required for the application, for example a keyboard interrupt handler to interpret keyboard input.

The clock interrupt handler provides the time slicing for the kernel. It knows when a process has used up its allotted time period. In this case the current process is preempted if any other processes are in the Ready queue. This interrupt handler also provides the mechanism for managing the Sleepqueue.

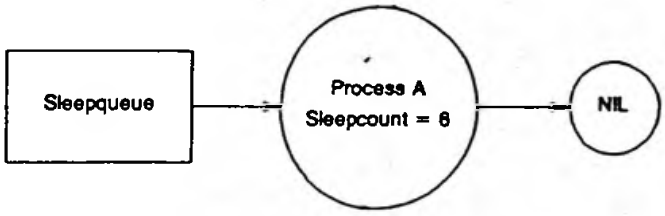
3.2.1 Clock Interrupt The clock interrupt handler is a PROCESS. On the IBM PC the clock interrupt occurs eighteen times each second. Each time the interrupt occurs the instructions of the interrupt handler are executed. The clock handler counts the number of clock pulses (clock interrupts) that have occurred since the current process began execution. When the count reaches the limit, the process is timed out. The clock handler places the current process at the end of the Ready queue. It removes the top process of the Ready queue and makes it the Cp. Finally the clock handler executes IOTRANSFER(clkhandlerP,Cp^.Cor,Clkintvec), transferring control to the new current process.

3.2.2 General Timer Service The clock handler also manages the Sleepqueue. Every time the clock interrupt occurs the clock handler decrements the Sleepcount of the top process in the Sleepqueue. When the Sleepcount reaches zero the sleeping process is started by removing it from the Sleepqueue and placing it in the Ready queue.

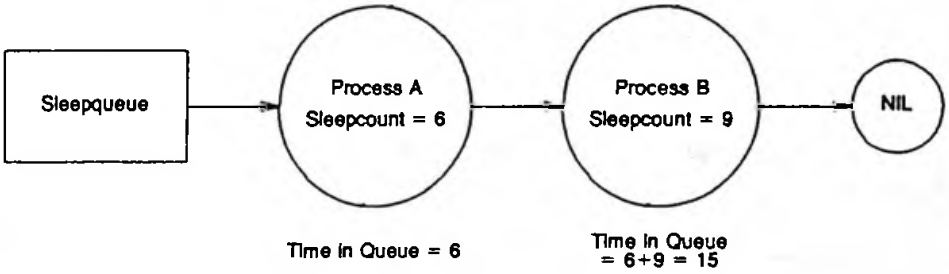
Insertions in the Sleepqueue are handled so only the Sleepcount of the top process need be decremented. Processes enter most Eventqueues by calling Wait; the process is inserted at the end of the queue. Sleepcount uses a different insertion algorithm.

To show how a process is inserted in the Sleepqueue we will follow the insertion of three processes in an initially empty queue, the first for six clock pulses, the second for fifteen and the third for ten.

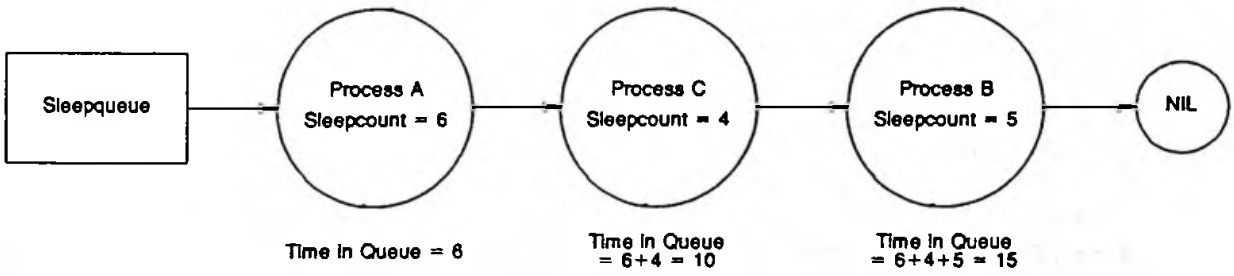
When the first process calls Sleep(6) it is placed at the top of the Sleepqueue with a Sleepcount of 6.



The second process calls Sleep(15). The parameter, 15, is compared with the Sleepcount of the first process. Since the second process is to sleep for a longer period, it is placed in the queue after the first process. The Sleepcount of the first process, 6, is subtracted from the parameter, leaving 9. Since there are no other processes, it is placed immediately after the first process with a Sleepcount of 9.



The third process calls Sleep(10). The parameter is compared to the Sleepcount of the first process in the queue. Since 10 is greater than 6, the third process will be placed behind the first. The Sleepcount of the first is subtracted from the parameter, leaving 4. This is compared to the Sleepcount of the next process of the queue, 9. Since 9 is greater than 4 the new process will be placed in the queue before this process. It is entered in the queue with a Sleepcount of 4 and the Sleepcount of the process immediately following is decremented by the Sleepcount of the new process, making it 5.



At this point there are three processes in the queue. The first process is the first in the queue, with a Sleepcount of 6. The third process is the second in the queue, with a Sleepcount of 4. The second process is the third in the queue, with a Sleepcount of 5. Each time a clock pulse occurs the Sleepcount of the top process in the queue is decremented. After 6 pulses the Sleepcount of the first process reached 0 and it is removed from the Sleepqueue and placed in the Ready queue. The third process is now the top in the queue, so each clock pulse its Sleepcount is decremented. After four clock pulses its Sleepcount is 0 and it is started. This leaves the second process. For five clock pulses its Sleepcount is decremented, it reaches 0 and the last process is started.

The first process slept for 6 clock pulses, the third for 6 + 4, or 10 clock pulses and the second for 6 + 4 + 5, or 15 clock pulses. All processes were inactive for the desired period of time and the updating required for each clock interrupt by the clock handler is reduced to updating and testing a single variable.

3.2.3 Countlock There is one other utility provided by the clock handler, the Countlock. By setting Countlock := TRUE a process can stop preemption; the clock handler is locked from timing the process out. The process has the processor until it executes Countlock := FALSE. This turns off the Countlock, allowing preemption.

There are tasks that can not be interrupted without the possibility of erroneous results. Countlock provides a simple mechanism to insure this does not happen. An example might be pointer operations on a linked list. Countlock is one way to insure the process is not timed out before the links have been properly rebuilt.

Kernel operations occur with the Countlock on. The Countlock can be invoked by the application as required. One such use of Countlock is the creation of primitive (i.e. may not be interrupted) instructions for semaphores.

3.3 Semaphores

A semaphore is a method used by two or more concurrent processes to synchronize access to a shared resource. A semaphore insures only one process has access to the resource at a time. It also insures that a process requesting the resource will eventually be allocated the resource.

There are two primitive instructions required for a semaphore. They are called P(s) and V(s) after the terminology used by E. W. Dijkstra. The operations required for P and V must be done without interruption if the results are to be guaranteed. Countlock enforces this restriction with clock interrupts by preventing time out pre-emption of the current process during the critical sections of P and V.

In the application presented here it was necessary to insure that only one process at a time used DOS. P and V were incorporated in the monitor that controlled access to DOS. Two variables, DOSinuse, a BOOLEAN and DOSqueue, an Eventqueue, are used. A process calls P before using DOS. The instructions used to implement P are:

```
Countlock := TRUE;
IF DOSinuse THEN
  Wait(DOSqueue)
ELSE
  DOSinuse := TRUE;
  Countlock := FALSE
END;
```

Flag DOSinuse is checked to see if another process has been allocated DOS. If not, flag DOSinuse is set to TRUE to show DOS is now allocated to the calling process and the process proceeds to use DOS. If DOSinuse is already TRUE when P is called, the calling program must wait until the process that allocated DOS, relinquishes it. It waits in Eventqueue DOSqueue until signalled.

When a process is no longer using DOS it calls V. The instructions used to implement V are:

```
Countlock := TRUE;
IF Awaited(DOSqueue) THEN
  Signal(DOSqueue)
ELSE
  DOSinuse := FALSE;
  Countlock := FALSE
END;
```

The process first checks to see if any process is waiting to use DOS. If so, that process is signalled; it can now resume execution and use DOS. DOSinuse is not reset to FALSE because DOS is now allocated to the signalled process. If no process is waiting for DOS, DOSinuse is set to FALSE and DOS is now free to be allocated to the next process calling P.

Without the Countlock, P and V may generate incorrect results. For example, Process A wishes to use DOS and calls P. It tests DOSinuse and finds it is FALSE. It is then timed out. Process B wishes to use DOS. It too calls P and it too finds DOSinuse to be FALSE as Process A was timed out before it could reset the flag. Process B sets DOSinuse to TRUE and proceeds to use DOS. While doing so it too is timed out. Process A resumes execution. It has already tested DOSinuse and still thinks DOSinuse is FALSE. It too sets DOSinuse to be TRUE and proceeds to use DOS. Because Process A was interrupted between testing and setting flag DOSinuse the semaphore has failed and two processes have been allocated DOS at the same time.

With the Countlock on, pre-emption by the kernel is impossible and the semaphore can not fail. In this case Countlock is combined with kernel procedures to create a higher level tool.

3.4 Limitations

The kernel has some limitations. The limitations do not prevent the kernel from being used for other applications. Any plans to use the kernel for other applications must take these limitations into consideration.

- There is no way to have priority access to resources. This means there is only a simple type of mutual exclusion. A process blocks another process from access to a critical section by blocking all other access to the CPU. This can cause performance problems when the time spent in a critical section is long.
- There is no way for a lengthy interrupt process to lower its priority so other interrupts may occur. This means interrupt processes must be designed carefully. They may not require much CPU time to execute nor may they cause the current application process to be pre-empted by another application process.
- Processes are scheduled using a simple round robin algorithm.
- The hardware clock interrupt is 17 Hertz. For many applications this does not provide a small enough interval for time outs or shared access to the system with time slicing.
- The amount of CPU time needed to switch from one process to another is higher than is necessary (the time to do a TRANSFER). The overhead of this procedure is from the need for the IOTRANSFER procedure to be inside a loop. This extra overhead will become less important as processors become faster. Interactive applications do not switch from interrupt processes to other processes with enough frequency for this overhead to be a problem. Other applications that do process frequent interrupts must consider the amount of CPU time needed to switch from one process to another.

4. *The Application System*

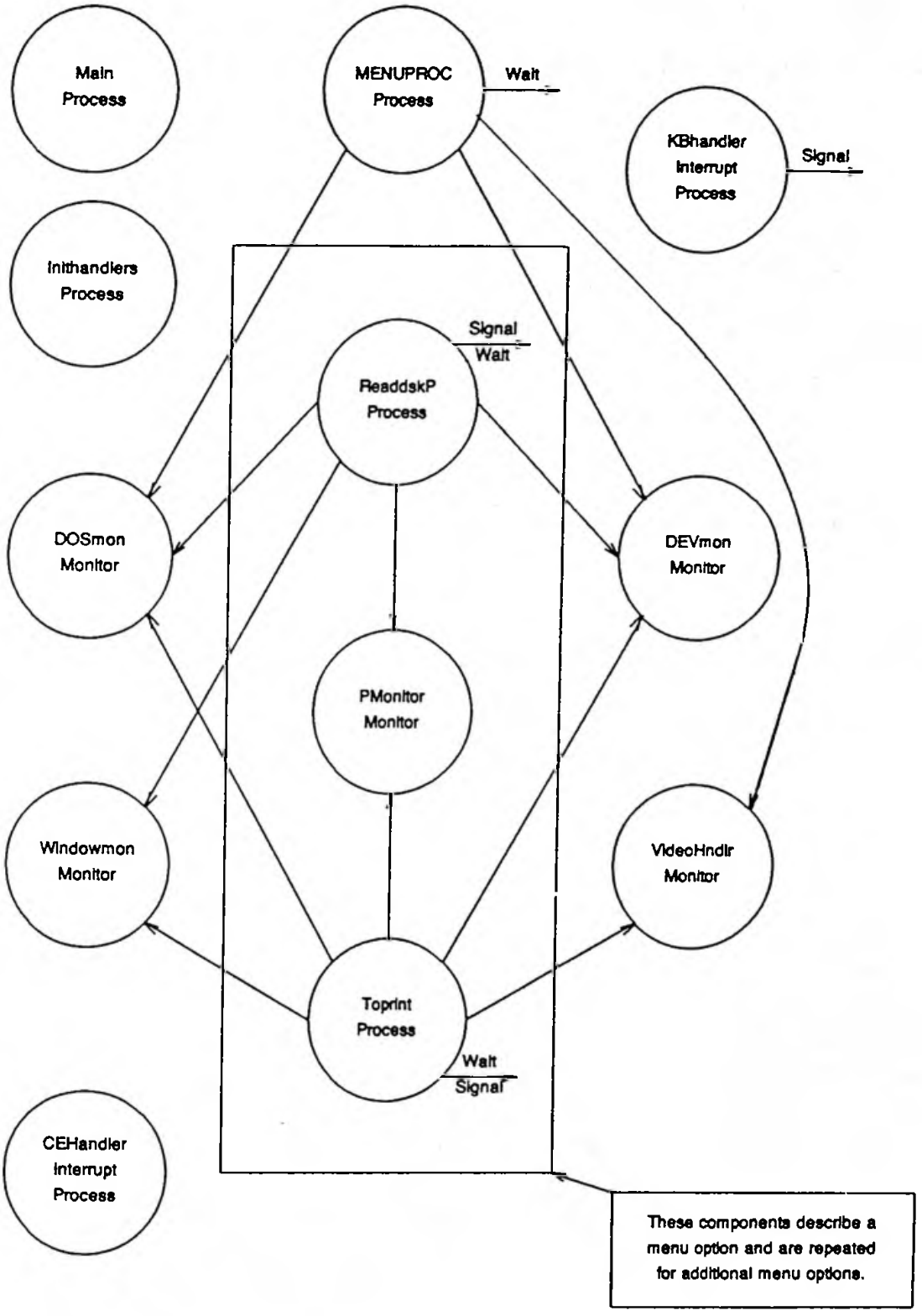
The Interactive Data Transfer System (IDTS) is an application system using the kernel routines to program simultaneous file transfers between various devices on the IBM PC. Files may be copied from disk to printer, disk to screen, and disk to disk. A split screen holds the menu and keyboard I/O on the top half of the screen.

Keyboard I/O and file transfers are all concurrent operations. While one file is writing to the printer, another may be writing to a disk, and a third file may be writing to the screen. The menu remains active during all file transfers so the user may type further requests at any time. File transfer requests for active output devices are held in queues and processed as devices become free.

Concurrency in the IDTS is handled by coroutines, called processes in Modula-2. True concurrency occurs when two or more processes are being executed by two or more processors at the same time. The processes in the IDTS are executed by the single processor of the IBM PC in quasi-concurrency. All active processes are given their share of running time by the Kernel's Scheduler routines.

4.1 The Application Design

The application system can be drawn as follows:


Main Process
MENUPROC Process
Wait
KBhandler Interrupt Process
Signal
Inithandlers Process
Signal
Wait
ReaddskP Process
DOSmon Monitor
DEVmon Monitor
PMonitor Monitor
Windowmon Monitor
VideoHndlr Monitor
Toprint Process
Wait
Signal
CEHandler Interrupt Process
These components describe a menu option and are repeated for additional menu options.

4.2 System Initialization

When a user starts the IDTS several things happen before the menu is active.

During Modula-2's module body initialization phase three interrupt handlers are installed by the process Inithandlers in the module Intrupts. The Scheduler uses Startprocesses to place Inithandlers in the Ready queue. The job of Inithandlers is to install the Clock interrupt handler (Timeout) used by the Scheduler, and the Keyboard interrupt handler (KBhandler) used by the Menu. Inithandlers also installs a simple Critical Error interrupt hander (CEHandler) that causes the IDTS to break and return to DOS on a fatal error. The screen is cleared and the menu itself appears on the top half of the screen. When the module body initialization is complete the main program module, Main, calls Startprocess for the Menu process and for all the file transfer processes (ReaddskP, Toprint, ReaddskS, Toscreen, ReaddskD, Todisk). Finally Main calls Startsystem.

Now Inithandlers installs the three interrupt handlers and calls Finishprocess. Inithandlers then disappears. Storage space is freed for other processes. Menu and the file transfer processes all Wait in Eventqueues until they are called into action by the appropriate Signal.

The Keyboard interrupt handler responds to a hardware interrupt on the IBM PC. It is enclosed within a high priority module so the keyboard input handling will be protected from interference from other processes. The handler routine responds to every key pressed by putting the key scan code into a simple buffer. Then it Signals to the Menu that there is a scan code available in the buffer. When this is finished the keyboard handler IOTRANSFERs back to the process it interrupted.

The module KBHandler contains and exports procedures for reading the buffer and interpreting the key scan codes. There are also procedures for a clean start and end of the interrupt handler in this module.

4.3 The Menu Process

The Menu process manages the application system.

MENUPROC (Menu Process) reads and processes key scan codes from the keyboard buffer using procedures from KBhandler. It echoes keyboard input on the menu (top) half of the screen. Menu manages cursor control and printing effects (e.g. reverse video) in the menu window with procedures from the module VideoHndlr (Video Handler).

The Menu process is also responsible for interpreting keyboard/menu input to handle file transfer requests. When a request is acceptable MENUPROC starts file transfers using the module DEVmon (Device monitor) for device request queue management. MENUPROC puts the requested file name in the correct device queue and Signals to start the Read process for the device. The Read process itself Signals to start the Write process for the device. For example, if the user has selected to print a file on the printer, MENUPROC Signals PrinterFreeIn, an Eventqueue for the Read-for-printer process, ReaddskP. The process is put in the Ready queue. ReaddskP in turn Signals PrinterFreeOut, an Eventqueue for the Write-to-printer process, Toprint. Now both processes are in the Ready queue where their execution time is managed by the Kernel.

File transfers are handled by three separate modules per device:

1. A module with a process that reads from the disk and writes to a buffer (e.g. READDSKP).
2. The buffer and its read and write procedures, Put and Get, are safe in a monitor (e.g. SMonitor) module that is only accessed by the two processes that need it.
3. A module with a process that reads from the buffer and writes to the output device (e.g. TOPRINT).

Eac Read/Write pair of processes run concurrently by calling Signal and Wait in the Kernel. These pairs of processes are made more loosely coupled by using Dijkstra's "Sleeping Barber" algorithm.

When a file transfer is complete (EOF) both processes check to see if any more requests are left in their device queue. If the queue is not empty both processes Wait, then Signal each other back into action. A user may request that the IDTS stop when all jobs are complete. When a process finds its queue empty and all other jobs complete, it stops the IDTS and exits to DOS. Otherwise, each process Waits in its Eventqueue for a Signal.

4.4 Keyboard and Display Device Interface

4.4.1 I/O Management The IDTS has several other monitors besides the file transfer monitors PMonitor (Printer monitor), SMonitor (Screen monitor), and DMonitor (Disk monitor).

IBM PC DOS is not reentrant. A process using DOS must be protected from other processes that also need DOS. Any process that uses DOS, either directly, (using the DOSCALL statement in Logitech Modula-2), or indirectly, (e.g. using file commands, disk access, or writing to the screen), uses DOSmon (DOS monitor). DOSmon protects DOS by only allowing one process at a time to access DOS. Whenever a process calls DOSmon it sets a BOOLEAN, DOSinuse, to TRUE. All other processes that request DOS are put in an Eventqueue by DOSmon, using Signal and Wait from the Scheduler.

DEVmon manages device requests. It insures only one file at a time is being written to a given device. DEVmon contains and manages the device request queues. A user may request file transfers on already busy devices. These will be started when the current transfers are completed. File requests are kept in first come first served queues.

4.4.2 Split Screen Functionality Windowmon (Window monitor) is a similar monitor that protects the lower screen windows. Any process writing to these windows, using the procedure WRITELine from VideoHndlr, must call Windowmon to insure only one process at a time has access to the window procedure.

The module VideoHndlr manages split screen output. File writes to the screen appear on the lower half of the screen and do not interfere with the menu nor with keyboard output on the top half of the screen. There are procedures in VideoHndlr that handle cursor positioning, writing to a window, and scrolling within a window. VideoHndlr also has procedures for the operations of ClearScreen, ReverseVideo, as well as Get/Set the Video mode.

VideoHndlr uses DOS and ROM BIOS routines for these operations. This module makes extensive use of low level facilities in Logitech Modula-2 such as DOSCALL, GETREG, SETREG, CODE, and SWI.

5. *Summary and Conclusions*

An Interactive Data Transfer System was built using a design structure composed of three programming components. It manages concurrent file transfers between three separate devices on an IBM PC system. Files may be transferred from a disk to a printer, a disk to the screen, and a disk to the same or another disk. The menu and keyboard I/O operations execute concurrent with file transfer requests. Users may make file transfer requests at any time. Transfer requests for an already active device are queued and processed when the device becomes free. The menu, messages, and output from disk files appear simultaneously on the screen in separate windows.

The application was designed by organizing the required system functions into processes, classes, and monitors. Processes were programmed as coroutines that run quasi-concurrently to handle file transfers and menu operations. Monitors were used to protect data, such as file buffers, and to provide mutual exclusion for operations that were not to be interrupted, such as calls to DOS routines that are not reentrant. The Kernel routines were used to manage (synchronize) all quasi-concurrent operations started by hardware interrupts.

Most components of the kernel and application were written using standard Modula-2. It was necessary to use tools provided with Logitech's Modula-2 to interface with low-level facilities such as DOS and ROM BIOS routines. Calls to DOS, ROM BIOS interrupts, and the inclusion of machine instructions are handled without writing separate assembly language routines. Those operations that were specific to the IBM PC, DOS, and the ROM BIOS exist in a few places and could be changed to work with Modula-2 compilers for other computer systems.

Modula-2 has proven itself to be an excellent language for building interactive applications that require system programming. Separate Definition modules from Implementation modules guarantees a clearly defined interface exists between software components. The separate compilation of modules prevents unnecessary recompiling of procedures that have not changed. When Modula-2 was used in a class project, students familiar with Pascal were able to learn the language. They were then able use it to complete a complex project in less than ten weeks of part time effort.

6. *Obtaining the Software*

A copy of the source files for the Modula-2 software described in this article may be obtained on a 48/TPI IBM PC floppy disk. To obtain the "Modula-2 Kernel & Application" software send a check or money order for $15.- (US Dollars) payable to:

The Association for Computing Machinery, Student Chapter
C/O Computer Science Department
San Francisco State University
1600 Holloway Avenue, Room TH 906
San Francisco, California 94132 USA

7. *Acknowledgements*

We want to thank those people who assisted with building the software and with the preparation of this paper: John Barr and Phil Rosine of Montana State University, in Missoula, Montana, for the first version a kernel; Jeff Clymer for his help during the programming of the class project; Christopher R. Cale, Maurizio Gianola, Alfred Moertlseder, of Logitech, Inc., in Redwood City, California, for donating Modula-2/86 software and technical help; John Copeland, Howard Ensler, and Brian Hart, of Image Network, in Mountain View, California, for providing technical help and access to the laser printers and typesetting software used to prepare this paper.

8. *Glossary*

analysis — The method of determining the essential components of a system, their features, and their relation to the other components of the system. Analysis is used to find the right combination of components to use when building a new system.

class — A program or program component that is a repeated operation.

concurrent — When two or more programs execute at the same time. These programs are possibly working together on a common activity.

coroutine — Two procedures or programs that are designed to work together. Each transfers control to the other as an intermediate processing step is completed. The software kernel of a computer system is a coroutine with all programs executing in the computer.

design — The plan or structure of an object. This is often a drawing or sketch that includes a written description of how to build the object and a list of the parts needed to build the object.

documentation — A written message that furnishes useful information about a system, or a component of a system to people.

event — Something important that occurs.

event queue — A data structure that contains a queue of processes waiting for an event to occur.

firmware — A programmable part of a system that is not easy to modify. A firmware program is frequently stored in memory that is read only. (Read/only memories are not easy to modify or to replace.) The machine running the program is often designed to run only one program. Firmware programs are hardware to a system designer who is unable to change the memory contents.

hardware — The physical part of a system. Hardware is used to build the machine components of a system. The hardware part of a computer system is not as easy to change as the programs that are stored using the hardware.

interrupt — A break in the execution of the currently running process of a machine. An interrupt is caused by a hardware device. The current process is stopped and a special process for the hardware causing the interrupt is started.

kernel — A set of programs and data structures that synchronize processes competing for use of machines. This allows one or more machines to be shared by the processes wanting to use the machine. The kernel is a monitor that controls the sharing of a machine. It is the part of a system that shares one or more machines between many processes.

machine — A part of a system that behaves in a predictable way and can not be changed during the lifetime of the system. A machine is frequently controlled by a list of instructions called a program.

methods — The instructions to people that allow machines to be used for useful tasks. These instructions are communicated to people in many forms. Usually this is done with documentation, and (or) through the supervision and control of an experienced manager.

monitor	A component of a system that supervises and manages the sharing of a resource between two or more processes.
operating system	A system that is working correctly. See "system".
people	Humans are one component of a system. To do useful work people follow methods for using machines.
preempt	When one process gains access to a resource before the process controlling the resource has finished using the resource. A system kernel uses this to allow the processes competing for a machine to share the machine.
procedure	A program that can be executed by other programs. A procedure can not execute concurrent with its calling program.
process	A list of programmed instructions for a machine that achieve a well defined result. A machine may execute only one process at a time. A program is built from one or more processes.
processor	A machine that executes a list of instructions.
program	A list of instructions for use by a machine. A program is built from one or more processes. When a program is stored in read/write memories it is called "software". When a program is stored in read/only memories it is called "firmware".
quasi-concurrent	When many processes appear to run concurrently with each other. This exists when there is only one machine to execute all the processes of a system. An interrupt preempts the current process to start the execution of another process.
real time	The ability of a process to be started and executed completely during a specific time period.
reentrant	A procedure that can be executed by one program while the procedure is being executed by other program. The instructions of the procedure are shared. Each program calling a reentrant procedure has a copy of the data used by the procedure.
semaphore	A way to signal between two concurrent (or quasi-concurrent) processes. A semaphore can be built from a data structure and two functions that use the data structure. The functions define "Signal" and "Wait" operations that allow one process to wait for the the signal from a second process.
software	A programmable part of a system that is easy to modify. A software program is frequently stored in memory that is read/write. (Read/write memories are easy to modify.) The machine running the program is often designed to run many different programs. Programs stored in a read/write memory are hardware to a system designer who is unable to change the memory contents.
system	People, Machines, and Methods working together for a useful purpose. A version of a system exists for a specific time period before it is modified, stopped or replaced.

9. References

[BAR78] Barr, J., *A Methodology for the Design of Interactive Graphics Operating Systems*, Computer Science Ph.D. Dissertation, University of California, Los Angeles, 1978.

[BEN82] Ben-Ari, M., *Principles of Concurrent Programming*, Prentice/Hall International, 1982.

[BRI74] Brinch Hansen, P., *The purpose of Concurrent Pascal*, Information Science, California Institute of Technology, November 1974.

[BRI75] Brinch Hansen, P., *Concurrent Pascal Report, Information* Science, California Institute of Technology, June 1975.

[BRI83] Brinch Hansen, P., *Using Personal Computers in Operating System Courses*, Computer Science Department, University of Southern California, June 1983.

[DEI84] Deitel, H. M., *An Introduction to Operating Systems*, Addison-Wesley Publishing Company, Reading, Revised First Edition, 1984.

[DEM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

[DIJ68] Dijkstra, E. W., "Cooperating Sequential Processes", in F. Genuys (ed.) *Programming Languages*, Academic Press, New York 1968, pp 43-112.

[DIJ71] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes", *Acta Informatica*, 1:115-138, 1971.

[DIJ73] Dijkstra, E. W., *Operating Systems Principles*, Prentice-Hall, New York, 1973.

[HOA74] Hoare, C. A. R., "Monitors: an operating system structuring concept", *Communications of the ACM*, 17 (10), 1974, pp 549-557 .

[HOA85] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice/Hall International, London, 1985.

[IBM85] *IBM PC DOS User's Manual*, Revised Editions, International Business Machines, Boca Raton, Florida, 1984.

[IBM85] *IBM PC Technical Reference Manual*, Revised Edition, International Business Machines, Boca Raton, Florida, April 1984.

[KIN83] King, R. A., *The IBM PC-DOS Handbook*, Sybex, Berkeley, California, 1983.

[LOG85] *Modula-2/86 User's Manual*, Release 2.00, Logitech, Inc., Redwood City, California, December 1985.

[NOR85] Norton, P., *The Peter Norton Programmers Guide to the IBM PC*, Microsoft Press, Belleveu, Washington, 1985.

[WIE84] Wiener, R. S., and Sincovec, R. F., "Modular Software Construction and Object-Oriented Design Using Modula-2", *Journal of Pascal, Ada, and Modula-2*, Vol. 3, No. 3, May/June 1984.

[WIR77] Wirth, N., "Modula: a Language for Modular Multiprogramming", *Software-Practice and Experience*, Vol. 7, pp 3-35, 1977.

[WIR85] Wirth, N., *Programming in Modula-2*, Third, Corrected Edition, Springer-Verlag, Heidelberg, 1985.

Dr. A. Brunnschweiler
Kehlhofstrasse 29
9322 Egnach
Switzerland

Mr. R. Karpinski
6521 Raymond Street
Oakland, CA 94609
U.S.A.

Egnach, November the 6th 1985

Dear Sir,

I greatly appreciate MODUS as a source of information on Modula-2. Since I am as interested in seeing some actual Modula-2 code, as some other readers seem to be, I should like to make a small contribution in this direction. Please feel free to throw it away or to print it, as you think best.

Having done some Pascal programming with the UCSD System, I was annoyed by the fact that Logitech Modula-2 does not support the Type SET OF CHAR. Of course this is not requested by Prof. Wirth, who states that the maximum number of elements of a set can be small and is machine dependent. But I have made extensive use of it in Pascal for bullet proof input and did not want to miss it. For this reason I have written a Module implementing sets of up to 256 characters. It is very simple to represent large sets as arrays of small sets, and to do all set operations, once the sets are defined. It is a bit more difficult to define such large sets in an elegant way. To be precise the problem is to do the equivalent of a Pascal statement like "S:=['a'..'e','A'..'E','.'];". My approach was to pass the information needed to define a set, as a string to a procedure DefineCharSet, which returns the set. Inside this procedure does the sort of things, which are usually done by a compiler. Thus the analogon to the above Pascal statement becomes: "DefineCharSet('(a)..(e),(A)..(E),(.)',S,ErrorNumber);".

With kindest regards

A. Brunnschweiler

P.S.

The actual code is on the included MSDOS 2.11 disk, written on an IBM compatible NCR PC-4i. Modus.def is the definition module and Modus.mod the implementation module.

```
DEFINITION MODULE ModusCharSet;

(*   ModusCharSet implements the Type SET OF CHAR, which usually
     is not available in Modula-2.

     A. Brunnschweiler  Kehlhofstrasse 29  9322 EGNACH  Switzerland.
     December the 6th. 1985
*)

EXPORT QUALIFIED CharSet, InCharSet, CharSetUnion, CharSetDifference,
   CharSetIntersection, CharSetSymmetricDifference, DefineCharSet;

TYPE CharSet = ARRAY[ 0..15 ] OF BITSET;

(* A 16 bit machine is assumed.  A set of 256 characters is implemented. *)

PROCEDURE InCharSet(CH: CHAR; S: CharSet): BOOLEAN;
      (*  InCharSet is true if and only if CH belongs to S.   *)

PROCEDURE CharSetUnion(A, B: CharSet; VAR C: CharSet);
      (*  C becomes the union of A and B.  *)

PROCEDURE CharSetDifference(A, B: CharSet; VAR C: CharSet);
      (*  C becomes the set difference of A and B  (C = A`B).  *)

PROCEDURE CharSetIntersection(A, B: CharSet; VAR C: CharSet);
      (*  C becomes the intersection of A and B.  *)

PROCEDURE CharSetSymmetricDifference(A, B: CharSet; VAR C: CharSet);
      (*  C becomes the symmetric difference of A and B.  *)

PROCEDURE DefineCharSet(Constructor: ARRAY OF CHAR; VAR S: CharSet;
                        VAR ErrorNumber: CARDINAL);

(*  Defines the set of characters S by means of Constructor.

    The syntax of Constructor in EBNF notation is:

    Set           = ElementarySet {',' ElementarySet}
    ElementarySet = [ PointSet [ '..' PointSet ] ]
    PointSet      = '(' CHAR ')' | '[' Number ']'
    Number        = [ Digit ] [ Digit ] Digit

    The meanings of PointSet and ElementarySet are:

    PointSet

    a set with one element.  As an instance both (0) and [48] denote
    the character '0', if the ordinals follow the ASCII convention.

    ElementarySet

    a simply connected set of characters.  As an instance [49]..[55] or
    (1)..(7) or [49]..(7) or (1)..[55] all denote the set of all characters
    between '1' and '7'.

    Remark: blanks, which are not enclosed in parentheses are ignored.
```

```
   Examples:

   DefineCharSet('(a), (f), (r)..(z)', S, ErrorNumber) defines S as
   the set containing the characters 'a', 'f' and all characters
   between 'r' and 'z'.

   DefineCharSet('[48]..[57], ( ), (A)..(Z), (.)', S, ErrorNumber) defines
   S as the set of all numerals, blank, all upper case characters and
   the point.

   ErrorNumber reports errors which occur when Constructor does not
   follow the syntax, given above.

    1)   Extra garbage in Constructor.
    2)   Two points are required to separate PointSets when building
         an ElementarySet.
    3)   Unexpected end of Constructor.
    4)   ) Expected.
    5)   ] Expected.
    6)   Digit expected.
    7)   Number too large.
*)

END ModusCharSet.
```

[Reformatted for ease of reading by the editor.]

```
IMPLEMENTATION MODULE ModusCharSet;

PROCEDURE InCharSet(CH: CHAR; S: CharSet): BOOLEAN;

VAR ActPosition,
    ActGroup:   [0..15];
    Membership: BOOLEAN;

BEGIN (* InCharSet *)
   ActGroup    := ORD(CH) DIV 16;
   ActPosition := ORD(CH) MOD 16;
   Membership  := (ActPosition IN S[ ActGroup ]);
   RETURN Membership;
END InCharSet;


PROCEDURE CharSetUnion(A, B: CharSet; VAR C: CharSet);

VAR I: CARDINAL;

BEGIN (* CharSetUnion *)
   FOR I := 0 TO 15 DO
      C[ I ] := A[ I ] + B[ I ];
   END;
END CharSetUnion;
```

```
PROCEDURE CharSetDifference(A, B: CharSet; VAR C: CharSet);

VAR I: CARDINAL;

BEGIN (* CharSetDifference *)
   FOR I := 0 TO 15 DO
      C[ I ] := A[ I ] - B[ I ];
   END;
END CharSetDifference;


PROCEDURE CharSetIntersection(A, B: CharSet; VAR C: CharSet);
VAR I: CARDINAL;

BEGIN (* CharSetIntersection *)
   FOR I := 0 TO 15 DO
      C[ I ] := A[ I ] * B[ I ];
   END;
END CharSetIntersection;


PROCEDURE CharSetSymmetricDifference(A, B: CharSet; VAR C: CharSet);
VAR I: CARDINAL;

BEGIN (* CharSetSymetricDifference *)
   FOR I := 0 TO 15 DO
      C[ I ] := A[ I ] / B[ I ];
   END;
END CharSetSymmetricDifference;


PROCEDURE DefineCharSet(Constructor: ARRAY OF CHAR; VAR S: CharSet;
                        VAR ErrorNumber: CARDINAL);

VAR CH:                CHAR;     (* Presently examined character of Construct
    CharPointer,                 (* Pointer to CH
    I:                 CARDINAL;
    EndOfConstructor: BOOLEAN;

PROCEDURE GetNextChar;

(* GetNextChar loads CH with the next valid character of Constructor or
   with 0C.  EndOfConstructor is true if and only if CH = 0C.
*)

BEGIN (* GetNextChar *)
   REPEAT
      IF CharPointer > HIGH(Constructor) THEN
         CH := 0C;
         EndOfConstructor := TRUE;
      ELSE
         CH := Constructor[ CharPointer ];
         IF CH = 0C THEN
            EndOfConstructor := TRUE;
         ELSE
            INC(CharPointer);
         END;
      END;
   UNTIL (CH <> ' ') OR EndOfConstructor;
END GetNextChar;
```

```
PROCEDURE ElementarySet(VAR ErrorNumber: CARDINAL);

VAR Number,
    ESetStart,
    ESetEnd:   CARDINAL;

PROCEDURE PointSet(VAR ErrorNumber: CARDINAL);

PROCEDURE GetNumber(VAR ErrorNumber: CARDINAL);

BEGIN (* GetNumber *)
   IF (ORD(CH) >= ORD('0')) AND (ORD(CH) <= ORD('9')) THEN
      Number := ORD(CH)-ORD('0');
   ELSE
      ErrorNumber := 6;
      RETURN;
   END;
   GetNextChar;
   WHILE (ORD(CH) >= ORD('0')) AND (ORD(CH) <= ORD('9')) DO
      Number := 10 * Number + ORD(CH) - ORD('0');
      GetNextChar;
   END;
   IF Number > 255 THEN
      ErrorNumber := 7;
      RETURN;
   END;
END GetNumber;

BEGIN (* PointSet *)
   IF CH = '(' THEN
      IF CharPointer > HIGH(Constructor) THEN
         CH := 0C;
         EndOfConstructor := TRUE;
      ELSE
         CH := Constructor[ CharPointer ];
         IF CH = 0C THEN
            EndOfConstructor := TRUE;
         ELSE
            INC(CharPointer);
         END;
      END;
      IF EndOfConstructor THEN
         ErrorNumber := 3;
         RETURN;
      END;
      Number := ORD(CH);
      GetNextChar;
      IF CH <> ')' THEN
         ErrorNumber := 4;
         RETURN;
      END;
      GetNextChar;
```

```
    ELSIF CH = '[' THEN
       GetNextChar;
       GetNumber(ErrorNumber);
       IF ErrorNumber <> 0 THEN
          RETURN;
       END;
       IF CH <> ']' THEN
          ErrorNumber := 5;
          RETURN;
       END;
       GetNextChar;
    ELSE
       ErrorNumber := 1;
       RETURN;
    END;
 END PointSet;

 PROCEDURE AddToCharSet;
 VAR TempCardinal: CARDINAL;
     ActPosition,
     ActGroup:     [0..15];

 BEGIN (* AddToCharSet *)
    IF ESetStart > ESetEnd THEN
       TempCardinal := ESetEnd;
       ESetEnd      := ESetStart;
       ESetStart    := TempCardinal;
    END;
    ActGroup    := ESetStart DIV 16;
    ActPosition := ESetStart MOD 16;
    INCL(S[ ActGroup ], ActPosition);
    WHILE ESetStart < ESetEnd DO
       INC(ESetStart);
       IF ActPosition = 15 THEN
          ActPosition := 0;
          INC(ActGroup);
       ELSE
          INC(ActPosition);
       END;
       INCL(S[ ActGroup ], ActPosition);
    END;
 END AddToCharSet;
```

```
BEGIN (* ElementarySet *)
   IF NOT ((CH = ',') OR EndOfConstructor) THEN
      PointSet(ErrorNumber);
      IF ErrorNumber <> 0 THEN
         RETURN;
      END;
      ESetStart := Number;
      ESetEnd   := Number;
      IF CH = '.' THEN
         GetNextChar;
         IF CH <> '.' THEN
            ErrorNumber := 2;
            RETURN;
         END;
         GetNextChar;
         PointSet(ErrorNumber);
         IF ErrorNumber <> 0 THEN
            RETURN;
         END;
         ESetEnd := Number;
       END;
      AddToCharSet;
   END;
END ElementarySet;

BEGIN (* DefineCharSet *)
   (* Initialisation. *)
   FOR I := 0 TO 15 DO
      S[I] := {};
   END;
   ErrorNumber      := 0;
   CharPointer      := 0;
   EndOfConstructor := FALSE;
   GetNextChar;

   ElementarySet(ErrorNumber);
   IF ErrorNumber <> 0 THEN
      RETURN;
   END;
   WHILE CH = ',' DO
      GetNextChar;
      ElementarySet(ErrorNumber);
      IF ErrorNumber <> 0 THEN
         RETURN
      END;
   END;
   IF NOT EndOfConstructor THEN
      ErrorNumber := 1;
      RETURN;
   END;
END DefineCharSet;

END ModusCharSet.
```

[Reformatted for ease of reading by the editor.]

Modula-2 News Issue # 0 October 1984

Purposes, practices and promises for Modula-2 News
Revisions and Amendments to Modula-2, Niklaus Wirth
Specification of Standard Modules, Jirka Hoppe
Modula-2 in the Public Eye (a bibliography), Winsor Brown
Modus Membership list, by name
Modus members's addresses, by location
Modula-2 Implementation Questionnaire

Modula-2 News Issue # 1 January 1985

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

MODUS Quarterly # 2 April 1985

Editorial
Letter on the draft Modula-2 Library, T. Anderson
Letter to the Editor, Mark Emerson
Opaque Types in Modula-2, C. French & R. Mitchell
Dynamic Module Instantiation, Roger Sumner
The Linking Process in Modula-2, Jeanette Symons
Modula-2 Library Comments, Bob Peterson
Modula Compilers - Where to Get 'em, Larry Smith
Coding War Games Prospectus, Tom DeMarco
M2, An Alternative to C, M. Djavaheri, S. Osborne

MODUS Quarterly # 3 July 1985

Editorial & potpourri of mail
Letter re opaque types, Steve Endicott
Letter on language issues, Christian Hoffman
Some Thoughts on Modula-2 in "Real Time", Paul Barrow
Letter re "actual Modula-2 code", Raja Thiagarajan
RajaInOut: simple, safer, I/O for
Logitech/MS-DOS, R. Thiagarajan
Selection of Contentious Problems, Barry Cornelius
Expressions in Modula-2, Brian Wichmann
The Scope Problems Caused by Modules, Barry Cornelius
Corrections and additions to Modula-2 compiler list

MODUS Quarterly # 4 November 1985

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

MODUS Quarterly # 5 February 1986

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

MODUS Quarterly # 6 November 1986

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

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

Hints for contributors:

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

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

Send your submissions to:

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

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : _____

Affiliation : _____

Address : _____

Address : _____

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

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

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

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

Aline Sigrist
ERDIS SA
CH-1800 Vevey 2

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

Modula-2 Users' Association
MEMBERSHIP APPLICATION

Name : _____

Affiliation : _____

Address : _____

Address : _____

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

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

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

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

Aline Sigrist
ERDIS SA
CH-1800 Vevey 2

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

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

MODUS
c/o ERDIS SA
Case Postale 35
CH-1800 Vevey 2