

The MODUS Quarterly

Issue # 5

February 1986

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

CONTENT

Cover 2. MODUS officers and contacts directory

Page 1. Editorial

2. Exporting a Module Identifier, Barry Cornelius

5. Letter on multi-dimensional open arrays, Niklaus Wirth

6. Letter on DIV, MOD, /, and REM, Niklaus Wirth

8. BSI Accepted Change: Multi-dimensional open arrays,  
Willy Steiger

10. N73: NULL-terminated strings in Modula-2, Ole Poulsen

14. ISO Ballot Results re BSI Specifying Modula-2

15. Draft BSI Standard I/O Library for Modula-2, Susan Eisenbach

24. Portable Language Implementation Project:  
Design and Development Rationale, K. Hopper and W. J. Rogers

42. The ETH-Zuerich Modula-2 compiler for the Macintosh,  
Chris Jewell

50. NewStudio: Engineering a Modula-2 Application for the  
Macintosh, A. Davidson, H. B. Herrmann, E. R. Hoffer

Cover 3. Membership form to photocopy

Cover 4. Return address

Copyright 1986 by MODUS, the Modula-2 Users Association.  
All rights reserved.

Non-commercial copying for private or classroom use is permitted.  
For other copying, reprint or republication permission,  
contact the author or the editor.



## Directors of MODUS, the Modula-2 Users Association:

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

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

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

Heinz Waldburger  
CH-1699 Maracon  
Switzerland  
(021) 93 88 24

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

### Administration and membership:

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

Europe: Heinz Waldburger  
Postfach 289  
CH-8025 Zuerich  
Switzerland  
(021) 93 88 24

### Editor, MODUS Quarterly:

>> Problems? Missing an issue? <<

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

Contact your membership  
coordinator (see above).

### Publisher:

George Symons (see above)

### Publication schedule:

#### 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.

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



## Modula-2 Users' Association MEMBERSHIP APPLICATION

Name: \_\_\_\_\_

Affiliation: \_\_\_\_\_

Address: \_\_\_\_\_

Address: \_\_\_\_\_

State: \_\_\_\_\_ Postal Code: \_\_\_\_\_ Country: \_\_\_\_\_

Phone: (\_\_\_\_) \_\_\_\_ - \_\_\_\_\_ Electronic Addr: \_\_\_\_\_

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

Application as: **New Member** \_\_\_\_ or **Renewal** \_\_\_\_

Implementation(s) Used: \_\_\_\_\_

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

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

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

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

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

Postfach 289  
CH-8025 Zürich  
Switzerland

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



Modula-2 News # 0 October 1984

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

Modula-2 News # 1 January 1985

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

MODUS Quarterly # 2 April 1985

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

MODUS Quarterly # 3 July 1985

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

MODUS Quarterly # 4 November 1985

MODUS Meeting Report by Bob Peterson  
A Writer's View of Conf, Sam'l Bassett  
Concerns of a Programmer, Dennis Cohen  
Mods to Standard Lib, Nagler & Siegel  
Std Lib and Ext'n to Modula-2, Odersky  
Std Lib for Unix by Morris Djavaheri  
Impl of Std Lib for PC's, Verhulst  
M-2 Compilation and Beyond, Foster  
Modula-2 Processes, Roger Henry

MODUS Quarterly # 5 February 1986

Export Module Identifier, Cornelius  
multi-dimensional open arrays, Wirth  
DIV, MOD, /, and REM, Niklaus Wirth  
Multi-dimensional open arrays, Steiger  
NULL-terminated strings, Poulsen  
ISO Ballot Results re BSI Modula-2  
Draft BSI I/O Library, Eisenbach  
Portable Language Rationale, Hopper +  
ETH-Z Modula-2 for Macintosh, Jewell  
NewStudio: for Macintosh, Davidson +

MODUS Administrators supply single copies at \$5 US or 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 troff, Script and PostScript 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 your submission is permitted. Correspondence not for publication should be PROMINENTLY so marked.

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

I think  
than w  
You c  
pref  
dis



EDITORIAL

The MODUS Quarterly #5, February 1986

I think we have a great issue here. And more printable submissions than we have room for. Consider the next issue to be partly filled. You can help to fill the rest with worthwhile articles: send me one, preferably by electronic mail or on Stride, IBM PC or Macintosh floppy disk. See the inside front cover for addresses, dates and hints.

You may recall that we have a quarterly prize now for the best article and for the best suggestion. Roger Henry's article on processes wins him a year of MODUS Quarterly. You may have thought that your vote would not count, but this time it would have. I got about one vote for best article. Send in your vote on the best article in this issue now. You can include your suggestion and get good odds on the cost of your stamp. We only got one of those last time, too.

Bill Nicholls suggested that we upgrade the appearance of MODUS Quarterly by typesetting the full text in a consistent font. Since many of our articles arrive in camera ready copy, that may take some time. Even so, I noticed that it would not be too hard to strip in some typeset page numbers to replace the crude hand written ones used to date. Let me know how you like the change. What is your suggestion? It counts too. Bill gets his next year of MQ for free.

Last week I went to Stanford to hear Chuck Clanton talk about his experiences in studying user interface issues for new microcomputer software products. He videotapes novice users as they try to accomplish assigned tasks. Sometimes, he asks his subjects to keep him posted on their thinking as they go along. In every case, he finds serious problems with the user interface. Even experts make profound mistakes in anticipating the problems, at least two thirds of the time, per decision. Only actual experiment with the intended audience can tell what works and what does not.

I followed Chuck to his car and bent his ear about my approach to the problem. I would encourage application designers to develop their programs as interpreters, with the commands (or elements to compose into commands) as separate procedures. There will be a central core pertaining to the purpose of the product, but the user interaction should be separated. In the end, the details of both the commands and views offered to the user can be configured at time of use. While any language can be used to develop programs in this way, Modula-2 shines.

For example, there may be several screens/windows with different sets of commands or of information available. Leaving the layout and content of those panels flexible is not hard; they will need change! The costs of this flexibility will be recovered by the time the second revision is required, following the second round of end-user testing. A simple text file may be sufficient to describe the whole range of possible configurations. The choice between showing a specific value numerically or in a bar-graph, gauge, elevator-bar or some other way is easy to specify here too. Furthermore, since the elements of user interface are separated from the application, they can be reused later in other contexts. Has anybody tried it? What problems came up?

rhk



## Exporting a Module Identifier

Barry Cornelius

Department of Computer Science  
University of Durham  
Durham DH1 3LE United Kingdom

The Modula-2 Working Group of the British Standards Institution has been trying to come to terms with the scope rules of Modula-2. This paper discusses one particular problem on which the Working Group is seeking feedback from the Modula-2 community.

The version of the Modula-2 Report in the 1st and 2nd editions of Wirth's book "Programming in Modula-2" states:

If a module identifier is exported, then all identifiers occurring in that module's export list are also exported.

However, this sentence has been removed from the version of the Report that appears in the 3rd edition of Wirth's book.

Consider the following example:

```
...  
PROCEDURE p;  
  MODULE n;  
    EXPORT m1, m2;  
    MODULE m1;  
      EXPORT a;  
      VAR a : INTEGER;  
    END m1;  
    MODULE m2;  
      EXPORT QUALIFIED b;  
      VAR b : INTEGER;  
    END m2;  
  END n;  
BEGIN  
  (* body of p *)  
END p;  
...
```

Is it legal to refer to a, b, m1.a and m2.b within the body



of procedure p? Four proposals have been considered:

	a	b	m1.a	m2.b
Proposal 1	illegal	illegal	OK	OK
Proposal 2	OK	OK	OK	OK
Proposal 3	OK	illegal	OK	OK
Proposal 4	illegal	illegal	illegal	OK

Wirth's original definition of the language agrees with either Proposal 2 or Proposal 3 - I'm not sure which. However, I believe that many implementations of the language

+ Modula-2 agree with Proposal 3. The definition that Wirth gives in the 3rd edition of his book leaves out the sentence quoted earlier. In a letter to Jeremy Siegel dated 20th February 1985, Wirth states:

If a module identifier m is exported, then the identifiers which m exports become visible, if qualified by m.

So, proposal 1 agrees with this new definition of the language. Finally, proposal 4 is taken from a development version of a Modula-2 implementation.

The BSI Modula-2 Working Group is aware that changes to the language may result in failure of existing programs. It is reasonably satisfactory when a language change results in an existing program failing to compile. However, it is far from satisfactory if a language change causes a new meaning for an existing program.

Here is an example of a program whose meaning changes:

```

...
VAR a : INTEGER;
PROCEDURE p;
  MODULE n;
    EXPORT m1;
    MODULE m1;
      EXPORT a;
      VAR a : INTEGER;
    END m1;
  END n;
BEGIN
  a := 27
END p;
...

```



Suppose this program is currently compiled by a compiler that implements the scope rules as defined by Proposal 3. So the programmer is making use of the fact that the assignment to a within the body of p is a reference to the variable exported from m1. Now, suppose a standards body decides that Proposal 1 should be adopted in the definition of a Modula-2 standard. If the programmer decides to switch to a compiler that complies with the standard, the assignment to a will now be interpreted as an assignment to the variable a that belongs to the block that surrounds procedure p. It may not be obvious to the programmer that he needs to alter his program. I agree that this is an extreme example and it may be that there are very few programs like this in existence!

However, a decision has to be made in order to produce a Modula-2 Standard. I guess it is possible to put forward arguments to support each of these proposals. For example:

- o Proposal 1 should be adopted because this is how Wirth currently thinks the language should be defined.
- o Proposal 3 should be supported because this is the definition adopted by a large number of implementations that interpreted the original definition of the language.
- o Proposal 1 should be supported because it is a better definition than the others.

The BSI Modula-2 Working Group is interested in the views of the Modula-2 community. In particular:

- (i) Do you have any strong opinions on which proposal should be adopted?
- (ii) Which proposal is adopted by the Modula-2 implementations that you use?
- (iii) Do you know of any programs that you use which have a different meaning depending on which proposal is adopted, i.e., the alteration will not be detected at compile-time?

Please send your comments to me as soon as possible. You may find this electronic mail address useful:

Barry\_Cornelius%DURHAM.MAILNET@MIT-MULTICS.ARPA



Computer Sciences Laboratory  
Xerox Corporation  
Palo Alto Research Center  
Palo Alto, California 94304  
415 494-4415

XEROX

31. July 85

Mr. John D. O'Meara  
8900 42nd Ave. NE  
Seattle WA 98115

Dear Mr. O'Meara,

This is in reply to your observation that Modula-2 lacks open array parameters with more than a single index.

I thank you for your suggestion and agree with you that this is a serious handicap when dealing with matrices. Fortunately the extension of the open array concept to more than one dimension is straightforward and lies entirely within the framework of Modula's concept. In fact, the only change required in the Report affects the syntax, namely (1) the last line on page 160 (3rd Ed, page 156 in 2nd Ed) in *Programming in Modula-2* and (2) rule 78 on page 172 (3rd Ed, rule 82, page 168, 2nd Ed) changes from

FormalType = [ARRAY OF] qualident.

to

FormalType = {ARRAY OF} qualident.

The associated interpretation (semantics) is quite obvious. One might add a sentence in section 10.1 on formal parameters after the syntax, explaining that the number of "ARRAY OF"s in the formal parameter specification must be identical to the number of index expressions in the actual parameter; but I think it is actually quite superfluous.

Note that this does not include the changes in the text of my book preceding the Report; but there are similarly few, and it is not the ultimate language specification anyway.

If this is felt to be essential, I should be happy to include this generalization in the next edition of the book. But you should be aware that individual implementors will still be free to update their compilers or to leave them with a new "restriction".

Sincerely yours,

D. Wirth



**ETH****EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE  
ZÜRICH**Institut für Informatik  
Fachgruppe Computer-SystemeClausiusstrasse 55  
Durchwahlnummer 01 256 22 26  
Sekretariat 01 256 22 27Postadresse:  
Institut für Informatik  
ETH-Zentrum  
CH-8092 Zürich

1610

Dr. Richard Karpinski  
Editor, MODUS Quarterly  
MODUS, Modula-2 User's Assoc.  
6521 Raymond St.

Oakland, CA 94609 / U S A

January 17, 1986

Dear Dr. Karpinski,

Thank you for your letter! No, I don't cover the topics on array parameters and integer arithmetic elsewhere, and you are free to reprint it. Perhaps the memo on arithmetic could be expanded into the following note:

Integer Arithmetic

The definition of the DIV and MOD operators in Modula-2 (and most other languages) is a persisting problem, and I believe that the standardization group and implementors should consider it seriously. In the report,  $x \text{ MOD } y$  is undefined for negative  $y$ , and for negative  $x$  the definition is in contradiction with mathematics. This should not be!

The trouble is that most computers implement arithmetic on integers in the sense of Euler, whereas modulo arithmetic is the one that is relevant in mathematics and - I believe - in computing too. Unfortunately they differ. Since it hardly made much sense to define a programming language with basic operations differing from those of practically all available hardware, the definition of the MOD operator in Modula is valid for positive operands only, where the two arithmetics are the same.

Fortunately, there is now at least one computer available whose designers recognized the issue: the NS32000. It offers instructions for both arithmetics. Our new compiler reflects this situation, and I suggest to adopt this solution generally.

In Euler's arithmetic, let the quotient be denoted by  $Q = x/y$  and the remainder by  $R = x \text{ REM } y$ . If  $y \neq 0$ ,  $Q$  and  $R$  always satisfy the equation

$$Q * y + R = x$$

and either  $0 \leq R < y$  or  $0 \geq R > y$ . Operations are symmetric with respect to zero, i.e.  $(-x)/y = x/(-y) = -(x/y)$ .



Modulo arithmetic is based on the idea of equivalence classes of integers. Each class can be identified by a specific member, for example its least non-negative member. For a given modulus  $y > 0$ , we create  $y$  classes, each class containing all integers  $q*y+r$  for arbitrary  $q$ , and its least, non-negative, identifying member being  $r$ , where  $0 \leq r < y$ . Hence we define  $q = x \text{ DIV } y$ , and  $r = x \text{ MOD } y$ , satisfying the equations

$$q*y + r = x \text{ and } 0 \leq r < y$$

Examples:

$31/10 = 3$	$31 \text{ REM } 10 = 1$
$-31/10 = -3$	$-31 \text{ REM } 10 = -1$
$31 \text{ DIV } 10 = 3$	$31 \text{ MOD } 10 = 1$
$-31 \text{ DIV } 10 = -4$	$-31 \text{ MOD } 10 = 9$

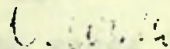
An example of the use of modulo arithmetic is the cyclic buffer of size  $n$  with indices in for the location of the next input and out for the next output. The number of filled slots is always  $(\text{in-out}) \text{ MOD } n$  (and not  $(\text{in-out}) \text{ REM } n$ ).

If negative integers are represented by their two's complement a right shift of  $x$  by  $k$  positions represents the operation  $x \text{ DIV } 2^k$  (and not  $x/2^k$ ).

(end of note)

Concerning multi-dimensional open arrays, I do not have anything further to say.

With best regards,



Prof. Niklaus Wirth

encl.



Date December 23, 1985  
From Logitech SA, Willy Steiger, member of BSI M2 Working Group

---

The following is a paper describing a change to the definition of Modula-2, as accepted by the BSI meeting of December 6 1985. I forward this to you so you can publish this decision in the next issue of MODUS newsletter, possibly together with other papers you may have received from one or the other member of the BSI M2 Working Group.

This modification introduces MULTI-DIMENSIONAL OPEN ARRAYS in Modula-2

### 1. Current Situation

---

Multi-dimensional open ARRAYS are currently not defined (cf. Report, CH.10.1). This limitation is felt to be a handicap for a number of applications, as numerics and statistics. The one-dimensional open ARRAY does exist, and the extension to more than one dimension is a natural one.

Note: Niklaus Wirth stated in a letter which was shown at the MODUS meeting in Palo Alto that multi-dimensional open ARRAYS would be an acceptable extension.

### 2. Accepted Change

---

Allow multidimensional open ARRAYS as parameters with the following syntax:

FormalType : { ARRAY OF } qualident.

This replaces the definition of 'FormalType' in chapter 10.1 of the report.

(The characters around the text 'ARRAY OF' in the above definition are curly brackets.) [ Curly brackets indicate possible repetition. rhk

### 3. Access to Multi-Dimensional Open ARRAYS Inside Procedures

---

In order to perform operations on the elements of open arrays, one must be able to know the number of elements in each dimension of the actual argument. For one-dimensional arrays the standard function HIGH(array) gives that value and an extension for more dimensions must be found. The obvious solution to introduce an optional second parameter, indicating the dimension, is rejected because (1) optional parameters are annoying for compiler writers and (2) the programmer would probably have to check in a manual to know if the first dimension is zero or one.

The solution that already applies for multi-dimensional fix arrays is usable for open arrays as well. Although this is not the most elegant solution, it has the merit not to alter the definition of the standard function HIGH. The principle is that the n-th element of a multi-dimensional array (more than n dimensions) is itself an array on which the function HIGH can be applied.



As with one-dimensional open ARRAYS, the array bounds of the formal parameters are mapped onto a CARDINAL-subrange starting from zero for all dimensions. The upper boundary of all dimensions can be obtained by using the existing standard function HIGH, as described in the following example:

---

```
MODULE OpenArray;
```

```
VAR array1, array2 : ARRAY [10..12],[0..3],[-1..11] OF CHAR;
```

```
PROCEDURE MatrixOp ( VAR m1, m2 : ARRAY OF ARRAY OF ARRAY OF INTEGER )
VAR
  i,j,k : CARDINAL;
```

```
BEGIN
```

```
(* the high boundary of an open array can be obtained like this: *)
```

```
  i := HIGH ( m1 ); (* first dimension *)
  j := HIGH ( m1 [ 0 ] ); (* second dimension *)
  k := HIGH ( m1 [ 0, 0 ] ); (* third dimension *)
```

```
(* or alternatively, loops can be directly controlled like this: *)
```

```
  FOR i := 0 TO HIGH ( m1 ) DO
```

```
    FOR j := 0 TO HIGH ( m1 [ 0 ] ) DO
```

```
      FOR k := 0 TO HIGH ( m1 [ 0, 0 ] ) DO
```

```
        m1 [ i, j, k ] := m2 [ i, j, k ] + 3;
```

```
      END (* third dimension *);
```

```
    END (* second dimension *);
```

```
  END (* first dimension *);
```

```
END MatrixOP;
```

```
BEGIN
```

```
  MatrixOP (array1, array2);
END OpenArray.
```

```
[ Reformatted and slightly revised by rhk. ]
```

---

Thank you for publishing this small piece of information about the work of the BSI Working Group for standardisation of Modula-2. This paper is submitted to you on behalf of that group.

Best regards,  
Logitech SA, December 23 1985



N73

N73:NULL-terminated strings in Modula-2.

-----

If null-terminated strings are to be regarded as useful, some generalisations / extensions to Wirth's definition in Programming In Modula-2 are necessary.

The following operations must be regarded as necessary parts of the Modula-2 language:

- 1) Assignment and comparison of string variables of different sizes.
- 2) Concatenation.
- 3) Use as parameters (actual as well as formal parameters), including the use of open array parameters.

1) Assignment:  $s1 := s2$

-----

Two cases must be considered:

a)  $SIZE(s1) \leq SIZE(s2)$

In this case, as much of  $s2$  as will fit in  $s1$  is copied.

b)  $SIZE(s1) > SIZE(s2)$

In this case,  $s2$  is copied to  $s1$  with a null character appended. See Figure 1.

-----

```
IF SIZE(s1) <= SIZE(s2)
  THEN FOR I := 0 to SIZE(s1) DO
    s1[I] := s2[I];
  END
ELSE I := 0;
  WHILE I < Length(s2) DO
    s1[I] := s2[I];
    INC(I);
  END
s1[I] := NULL;
END
```

-----

FIGURE 1.



N 73

Open entry parameters...

Comparison: s1 Relop s2

---

where Relop belongs to {<, <=, >, >=, =}

Comparisons should cause no problems, other than possible difficulties of implementing the operators. See Figure 2.

---

```
PROCEDURE LessThan(S1: string1; S2 : string2) : BOOLEAN;
VAR Index : CARDINAL;
BEGIN
  Index := 0;
  LOOP
    IF (Index > SIZE(s1)) OR (S1[Index] = CHR(0))
      THEN RETURN NOT ((Index > SIZE(s2)) OR
        (S2[Index] = CHR(0))) END;
    ELSIF Index > SIZE(s2) THEN RETURN FALSE;
    ELSIF S1[Index] < S2[Index] THEN RETURN TRUE;
    ELSIF S1[Index] > S2[Index] THEN RETURN FALSE;
  END;
  INC(Index);
END;
END LessThan;
```

---

FIGURE 2.

2) Concatenation: s3 := s1 + s2

---

This process is most easily described by looking at Figure 3.

In cases where the result of a concatenation is used as an actual parameter, the following applies:  $SIZE(s1+s2) = SIZE(s1) + SIZE(s2)$ .



```

-----
I := 0;
J := 0;

WHILE (J <= SIZE(s3)) AND (I <= SIZE(s1)) AND
      (s1[I] <> NULL) DO
    s3[J] := s1[I];
    I := I+1;
    J := J+1;
END;

I := 0;

LOOP
  IF (J > SIZE(s3)) THEN EXIT; END;
  IF (I > SIZE(s1)) THEN s3[J] := CHR(0); EXIT; END;
  s3[J] := s1[I];
  IF s2[I] = CHR(0) THEN EXIT; END;
  I := I+1;
  J := J+1;
END;
-----

```

FIGURE 3.

### 3) The use of a string variable as an actual parameter: P(s)

As shown in Figure 4, SIZE(s) must be equal to SIZE(x) where x is the formal parameter corresponding to s.

```

-----
TYPE T1 = ARRAY [0..9] OF CHAR;
VAR S1 = ARRAY [0..2] OF CHAR;

PROCEDURE P(VAR S : T1);
BEGIN
  S := '0123456789';
END P;

BEGIN
  P(S1);
END.
-----

```

FIGURE 4.



### Open array parameters.

---

In connection with string operations, open array parameters of the type ARRAY OF CHAR are compatible with ARRAY [0..n] OF CHAR where n is equal to the length of the actual parameter with the limitation that the use of a formal open array parameter as an actual parameter is permitted only if the corresponding formal parameter is an open array parameter.

### Characters relation to Strings.

---

With respect to strings, variables of type CHAR are compatible with ARRAY [0..0] OF CHAR. This means, for example, that the following is allowed:

---

```
VAR C : CHAR
PROCEDURE P(v : ARRAY OF CHAR);
BEGIN
END P;

BEGIN
  P(C);
END.
```

---

FIGURE 5.

Ole Poulsen  
Borland International (U.K) LTD  
Wicham House, 10 Cleveland Way,  
London, E1 4TR



N-78

## SPECIFICATION FOR THE COMPUTER PROGRAMMING LANGUAGE MODULA 2

A Letter Ballot (Attachment to 97/22 N076) was circulated to SC22 with a return date of 1985-10-15.

The following responses have been received:

'P' Members supporting proposal:	9 (Austria, Canada, China, France, Germany F.R., Italy, Sweden, UK, and USA)
'P' Members not supporting proposal:	2 (Japan and Netherlands)
'P' Members submitting comments:	4 (France, Japan, Netherlands, and USA)
'P' Members having abstained:	0
'P' Members not voting:	4 (Belgium, Finland, Norway, and USSR)

### Comments:

France	- Attachment A
Japan	- Attachment B
Netherlands	- Will follow
USA	- Attachment C

### Secretariat Action:

This proposal, having obtained sufficient support, will be forwarded by the SC22 Secretariat to the TC97 Secretariat for confirmation and circulation to the P-Members of TC97.



N-78

## Draft BSI Standard I/O Library for Modula-2

The following is an interim report on the progress of the standard I/O library.

### Design Goals

We wanted to create an I/O library which would be sufficiently general purpose for widespread use. We identified three properties:

- (i) It should be layered, to ease implementation and to allow different types of user access to features and facilities of varying levels of power and sophistication.
- (ii) It should be extensible so that facilities for user-defined data types and device-specific drivers can be added at will.
- (iii) It should be robust so that ordinary high-level language programmers can have error-free and crash-free access to the I/O devices without having to resort to doing their own parsing.

### Layering

The current state of the design sees three layers. The bottom level contains the device-specific driver modules. There must be a module for each device to be accessed through a Modula program.

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

The top level contains the type-specific modules. The definition modules below are those for this level only. There is one module for each of the standard Modula simple types. The same operations are available for each type. Details can be found in the commentary and definition modules below.

### Extensibility

When data of a specific type is to be input or output, the relevant I/O module must be imported. When new types are defined, the programmer can create a similar module to handle that type. Thus i/o operations are orthogonal across the range of types.

The library is perceived as being suitable for four classes of user.

- (i) Applications programmers using standard types (using the top level).
- (ii) Programmers creating new types for which they want efficient implementations of the I/O operations (using the middle level to extend the top level).
- (iii) Programmers directly controlling devices (using the bottom level).



- (iv) Programmers interfacing new devices to the system (extending the bottom level.

### Robustness

Using this system, it is straightforward to write programs that will not crash regardless of the actual input received from the channel. This is achieved by the use of predicates for pretesting to indicate whether a specific input operation will succeed. This is in contrast to the method where the I/O operation takes place and if it fails the programmer can take corrective action.

## **The Semantics Of the Top Level Interface**

### Introduction

This description of the semantics of the top level interface to the Modula I/O Library should be read in conjunction with the definition modules. The top level interface deals with the input and output of the five Modula standard base types, **CHAR**, **CARDINAL**, **INTEGER**, **BOOLEAN** and **REAL**. All input and output is performed via explicitly referenced channels, supported by the middle level interface that are connected to any device. Thus this top level **IMPORTS** from the middle level the opaque type Channel. Each of the five types have their own I/O module, all of which export 7 procedures. These procedures are identical in semantics but as they deal with different objects have slightly different syntax. The only exception to this is the Print procedure of the RealIO module which has an extra parameter. It is possible then to describe the semantics of all the modules by describing the semantics of any one of them, for this reason types are not explicitly referred to (except for examples) but are referred to as objects.

### Channels

Although channels are not described here, it is important to note one or two of their attributes which directly effect the semantics of the procedures to be described shortly. Firstly all channels use blocking reads, that is, if no data is currently available from the device attached to the channel, a blocking read is caused and the procedure hangs until data is available. Secondly, some channels may be opened as binary channels, in this case the implementation of the object I/O modules will ensure that the object is written in a binary form. The various procedures are now described.

### CanSkip

This predicate function returns **TRUE** if an object is available from the channel specified. The state of the channel is not altered. An object is deemed to be available if the channel is a character channel and the syntactic rules for that object can be followed. An object is also deemed to be available if the channel is a binary channel and sufficient data is available from the channel to form the object, the amount of data that will be required for an object will be system dependent (depending on the value of TSIZE(object)). If the channel has reached the end of stream condition



ing the  
(not available from this level of the I/O system) then CanSkip will return FALSE.

### CanRead

This predicate function returns TRUE if an object is available from the channel specified and it can be represented internally. The state of the channel is not altered. An object is deemed to be available if a call to CanSkip returns TRUE and the object, when interpreted, can be represented by the system. The result of this function, therefore, will be system dependent for the same stream of data from a character stream. A return of TRUE from this function implies that the value of the object represented by the character stream associated with the channel can be represented within the computers hardware. In the case of a binary channel, sufficient data must be available from the channel to form a valid object, as with CanSkip this will be system dependent. If the channel has reached its end of stream condition then CanRead returns FALSE.

### Skip

This procedure will skip over the current object on the channel, thus changing the state of the channel. As a precondition to the successful operation of the procedure the predicate CanSkip must be TRUE. If it is not then a fatal I/O error occurs.

### Value

Will return as a result of the procedure the value of the next object available from the specified channel. The state of the channel is not altered and so subsequent calls to Value will return the same result. As a precondition to the successful evaluation of the object, the predicate CanRead (See above) must be TRUE. If it is not then a fatal I/O error occurs.

### Read

Will place in its second parameter the current object on the channel. The predicate CanRead must be true else a fatal I/O error is caused. The object is then skipped, thus altering the state of the channel.

### Write

Will write its parameter to the channel. In the case of character channels the minimum space possible is used. Positive INTEGERS are unsigned, and REALs are output in standard form. In the case of binary channels, the object is written in binary coded form. The state of the channel is changed. If, for some reason, the channel cannot be written to, or the write fails, then a fatal I/O error is caused.

### Print

Will write its parameter to the channel in a formatted way. The use of print to a binary channel is illegal and will result in a fatal error. The number of character positions to be used is defined by the Length



parameter and the justification can be left or right. For RealIO a fourth parameter specifies the number of decimal places to be printed.

### Errors

In the event of an I/O error described above occurring, a fatal I/O Error is caused. If such an error occurs a message stating the type of error is displayed and the program is terminated. It is suggested that if a fail and recover I/O system is desired then control should be transferred to some explicit error handling procedure so that a user may attempt to recover from the error under program control.

### Conclusion

This is the top level of the proposed BSI Modula-2 I/O library. We hope to have the rest of the definition modules and their implementations by the next issue of the Modus newsletter. We are releasing these because we would like to get some feedback on the direction we are going.

Susan Eisenbach  
Department of Computing  
Imperial College  
180 Queens Gate  
London SW7 2BZ  
se@icdoc.ac.uk

DEFINITION MODULE IO ;

EXPORT QUALIFIED  
Channel ;

TYPE  
Channel ;

END IO.



fourth  
20  
DEFINITION MODULE BoolIO;

FROM IO IMPORT  
Channel;

EXPORT QUALIFIED

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

(\*

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

PROCEDURE CanRead (C: Channel): BOOLEAN;

PROCEDURE CanSkip (C: Channel): BOOLEAN;

PROCEDURE Value (C: Channel): CARDINAL;

PROCEDURE Skip (VAR C: Channel);

PROCEDURE Read (VAR C: Channel; VAR Bool: BOOLEAN);

PROCEDURE Write (VAR C: Channel; Bool: BOOLEAN);

PROCEDURE Print (VAR C: Channel; Bool: BOOLEAN; Length: CARDINAL;  
RightJustified: BOOLEAN);

END BoolIO.



DEFINITION MODULE CharIO;

FROM IO IMPORT  
Channel;

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

(\*

- \* If no data is currently available then the predicates wait for some.
- \*
- \* CanRead returns TRUE if the current object can be represented.
- \* CanRead implies CanSkip.
- \*
- \* CanSkip returns TRUE if there is an object available.
- \* CanSkip does not imply CanRead as an object could be well-formed but out of range.
- \*
- \* Value returns the current object.
- \* CanRead MUST be TRUE.
- \* If a call to CanRead currently returns FALSE then Value will fail.
- \*
- \* Skip skips over the current object.
- \* CanSkip MUST be TRUE.
- \* If a call to CanSkip currently returns FALSE then Skip will fail.
- \*
- \* Read places the current object into its second parameter and then skips over it.
- \* CanRead MUST be TRUE.
- \* If a call to CanRead currently returns FALSE then Read will fail.
- \*
- \* Write the object to the channel with no padding.
- \*
- \* Print the object to the channel (with padding). Length is the minimum number of characters that must be output. It is an error to Print to a binary channel.
- \*
- \* NOTE: The object is written in binary or character form depending on how the channel was opened.
- \*)

PROCEDURE CanRead (C: Channel): BOOLEAN;

PROCEDURE CanSkip (C: Channel): BOOLEAN;

PROCEDURE Value (C: Channel): CHAR;

PROCEDURE Skip (VAR C: Channel);

PROCEDURE Read (VAR C: Channel; VAR Ch: CHAR);

PROCEDURE Write (VAR C: Channel; Ch: CHAR);

PROCEDURE Print (VAR C: Channel; Ch: CHAR; Length: CARDINAL;  
RightJustified: BOOLEAN);

END CharIO.



DEFINITION MODULE CardIO;

FROM IO IMPORT  
Channel;

EXPORT QUALIFIED

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

(  
\* Numeric objects ignore all leading white space  
\* (Space, Tab, LF, CR etc) and are terminated by  
\* the first character that would be illegal in  
\* the object. This illegal character is left on  
\* the channel.  
\*  
\* If no data is currently available then the predicates wait for some.  
\*  
\* CanRead returns TRUE if the current object can be represented.  
\*     CanRead implies CanSkip.  
\*  
\* CanSkip returns TRUE if there is an object available.  
\*     CanSkip does not imply CanRead as an object could be well-formed  
\*     but out of range.  
\*  
\* Value returns the current object.  
\*     CanRead MUST be TRUE.  
\*     If a call to CanRead currently returns FALSE then Value will fail.  
\*  
\* Skip skips over the current object.  
\*     CanSkip MUST be TRUE.  
\*     If a call to CanSkip currently returns FALSE then Skip will fail.  
\*  
\* Read places the current object into its second parameter and then  
\*     skips over it.  
\*     CanRead MUST be TRUE.  
\*     If a call to CanRead currently returns FALSE then Read will fail.  
\*  
\* Write the object to the channel with no padding.  
\*  
\* Print the object to the channel (with padding). Length is the minimum  
\*     number of characters that must be output. It is an error to Print  
\*     to a binary channel.  
\*  
\* NOTE: The object is written in binary or character form  
\*     depending on how the channel was opened.  
\*)

PROCEDURE CanRead (C: Channel): BOOLEAN;

PROCEDURE CanSkip (C: Channel): BOOLEAN;

PROCEDURE Value (C: Channel): CARDINAL;

PROCEDURE Skip (VAR C: Channel);

PROCEDURE Read (VAR C: Channel; VAR Card: CARDINAL);

PROCEDURE Write (VAR C: Channel; Card: CARDINAL);

PROCEDURE Print (VAR C: Channel; Card: CARDINAL; Length: CARDINAL;  
                  RightJustified: BOOLEAN);

END CardIO.

DEFINITION MODULE IntIO;

FROM IO IMPORT  
Channel;

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

(\*

- \* Numeric objects ignore all leading white space
- \* (Space, Tab, LF, CR etc) and are terminated by
- \* the first character that would be illegal in
- \* the object. This illegal character is left on
- \* the channel.

\*)

- \* If no data is currently available then the predicates wait for some.

\*)

- \* CanRead returns TRUE if the current object can be represented.
- \* CanRead implies CanSkip.

\*)

- \* CanSkip returns TRUE if there is an object available.
- \* CanSkip does not imply CanRead as an object could be well-formed
- \* but out of range.

\*)

- \* Value returns the current object.
- \* CanRead MUST be TRUE.
- \* If a call to CanRead currently returns FALSE then Value will fail.

\*)

- \* Skip skips over the current object.
- \* CanSkip MUST be TRUE.
- \* If a call to CanSkip currently returns FALSE then Skip will fail.

\*)

- \* Read places the current object into its second parameter and then
- \* skips over it.
- \* CanRead MUST be TRUE.
- \* If a call to CanRead currently returns FALSE then Read will fail.

\*)

- \* Write the object to the channel with no padding.

\*)

- \* Print the object to the channel (with padding). Length is the minimum
- \* number of characters that must be output. It is an error to Print
- \* to a binary channel.

\*)

- \* NOTE: The object is written in binary or character form
- \* depending on how the channel was opened.

\*)

PROCEDURE CanRead (C: Channel): BOOLEAN;

PROCEDURE CanSkip (C: Channel): BOOLEAN;

PROCEDURE Value (C: Channel): INTEGER;

PROCEDURE Skip (VAR C: Channel);

PROCEDURE Read (VAR C: Channel; VAR Int: INTEGER);

PROCEDURE Write (VAR C: Channel; Int: INTEGER);

PROCEDURE Print (VAR C: Channel; Int: INTEGER; Length: CARDINAL;  
RightJustified: BOOLEAN);

END IntIO.



DEFINITION MODULE RealIO;

FROM IO IMPORT  
Channel;

EXPORT QUALIFIED

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

(\*  
\* Numeric objects ignore all leading white space  
\* (Space, Tab, LF, CR etc) and are terminated by  
\* the first character that would be illegal in  
\* the object. This illegal character is left on  
\* the channel.  
\*  
\* If no data is currently available then the predicates wait for some.  
\*  
\* CanRead returns TRUE if the current object can be represented.  
\*     CanRead implies CanSkip.  
\*  
\* CanSkip returns TRUE if there is an object available.  
\*     CanSkip does not imply CanRead as an object could be well-formed  
\*     but out of range.  
\*  
\* Value returns the current object.  
\*     CanRead MUST be TRUE.  
\*     If a call to CanRead currently returns FALSE then Value will fail.  
\*  
\* Skip skips over the current object.  
\*     CanSkip MUST be TRUE.  
\*     If a call to CanSkip currently returns FALSE then Skip will fail.  
\*  
\* Read places the current object into its second parameter and then  
\*     skips over it.  
\*     CanRead MUST be TRUE.  
\*     If a call to CanRead currently returns FALSE then Read will fail.  
\*  
\* Write the object in fixed format to the channel with no padding.  
\*  
\* Print the object to the channel in floating point format (with padding).  
\*     Length is the minimum number of characters that must be output.  
\*     It is an error to Print to a binary channel.  
\*  
\* NOTE: The object is written in binary or character form  
\*     depending on how the channel was opened.  
\*)

PROCEDURE CanRead (C: Channel): BOOLEAN;

PROCEDURE CanSkip (C: Channel): BOOLEAN;

PROCEDURE Value (C: Channel): REAL;

PROCEDURE Skip (VAR C: Channel);

PROCEDURE Read (VAR C: Channel; VAR R: REAL);

PROCEDURE Write (VAR C: Channel; R: REAL);

PROCEDURE Print (VAR C: Channel; R: REAL; Length: CARDINAL;  
RightJustified: BOOLEAN);

END RealIO.

## PORTABLE LANGUAGE IMPLEMENTATION PROJECT: DESIGN AND DEVELOPMENT RATIONALE

by  
K Hopper & WJ Rogers

*Abstract The practical reasons for starting to work on a portable compiler system are introduced as the basis for requiring transparent portability for compilation/running/testing of Modula-2 programs. The shortcomings of Modula-2 led to the design of a new system software language, Peano and the integration of the two compilers together with editing and debugging facilities into a single major software engineering project. The objective of producing plug-and-socket programs as an aid to portability has been achieved with the aid of structured intermediate languages for compiling, editing and interpreting. The main train of thinking for all major design aspects of the project is discussed in an attempt to give an understanding of its principles to those coming to this technique for the first time.*

### Introduction

It was realised late in 1983 that the teaching/learning of computer hardware courses could be considerably simplified by the use of a high-level language which would offer direct access to hardware facilities. This was to be used to allow students to program hardware laboratory experiments without the need to be experienced in a variety of assembler languages. Modula-2 was the high-level language chosen since it is closely related to the Pascal language learnt by students during their first year of study.

Two different Modula-2 implementations were obtained - for the PDP-11 and VAX-11 machines. After some initial problem installing them and getting them to work, it was soon discovered that there were significant differences - not only in the execution of programs accepted by both compilers, but also in the programs which would not be accepted by one and would be accepted by the other!

After a little more experience it was discovered that the ostensibly identical run-time systems accompanying the two compilers behaved rather differently under quite a wide variety of circumstances. All these experiences led to the realisation that while Modula-2 may be a good system programming language, the difference between different implementations made it of very questionable use in a teaching environment. Students would be using different compilers for different hardware and trying to differentiate their own errors from implementation variations - most undesirable.

At about the same time that consideration was being given to using Modula-2, planning for enhancement of hardware laboratory facilities was centring around the Zilog series of 16-bit microprocessors - for which no Modula-2 compiler yet existed! Since a new compiler would be needed for the microprocessors, it was decided to produce a Modula-2 system which would work identically on the university VAX machines and the Zilog Z8000 series microprocessors.

### Original Goals

Although every language implementer inevitably thinks of some new 'feature' or gimmick which would be useful for his own use, it was decided to be very conscientious in implementing exactly the language defined by Niklaus Wirth. The opportunity almost offered by the pseudo-module SYSTEM for fancy extras was to be avoided at all costs. It was decided that the only features other than those predefined by the language were to be those needed within the compiler system in order to implement the language in a machine-independent way!

The problems of inconsistent run-time support also had to be solved so that programs written without specific machine dependencies would compile and function identically on any machine available to students. This not only meant machine independence but, much more important, operating system independence too!

### Project Expansion

The inadequacies of Modula-2 as a system programming language are primarily in the areas of type-safeness and data abstraction. Relaxation of type-checking in Modula-2 is sudden and total. This is very frequently undesirable since in almost all practical cases only some minor relaxation is needed - not dispensation with the entire programmer protection system! The private data types offered by Modula-2 are restricted to being



pointer-sized types - rather than providing a perfectly general private type facility as offered, for example, by Ada. Unfortunately, while Ada is a very powerful language with strong type checking, it is not possible to provide a reasonably small implementation of the full language. For this reason it was decided that another new systems language would be most useful, particularly for smaller machines.

In view of the shortcomings already noted, it was decided that this language should be designed with the following philosophy

- a. Full type checking should be provided on all objects as a default.
- b. Relaxation of type-checking should be provided in a controlled way, it being possible to ignore individual aspects of type-checking on an object quite independently of other type-checking needs.
- c. The language should be functional, with all routines being niladic and all functions/operators monadic or dyadic - using infix notation.
- d. Operators may be overloaded in any way required by the programmer - short of ambiguity.
- e. Generic types/functions/routines and modules would be provided with facilities for instantiation in several stages as required.
- f. Modules may be either entire, a face or a body. Entire modules may be generic and may be declared at any point in a program.
- g. Every object must be first-class - it may appear at any point in an expression of appropriate type!
- h. The intermediate code may be written in-line if it is necessary to override automatic compiler allocation of resources or to ensure the use of some particular instruction or sequence which may (not) have desirable/undesirable side effects, etc. The language thus becomes on these relatively rare occasions a two-level one.

The new language is called Peano since it is defined axiomatically and allows other languages to be readily defined in terms of its objects and operations. In designing this language a number of strategic design decisions were made while attempting to satisfy the needs outlined above. These were principally aimed at providing what was subjectively felt to be a practically usable language within the general philosophy

- a. The language should compile rapidly. In practice this means a one pass compiler - only reading the source language once! This, in turn, requires the declaration of an identifier before it may be used and the provision of *forward* declarations (similar to those in Ada). The use of a forward-declared type in a type-safe language means that *NO* type-checking can be done until the full-declaration is encountered - a forward type must therefore be incompatible with every other type!
- b. No run-time support should be necessary in any implementation. The concepts of such things as heap management, input/output and concurrency have therefore been deliberately omitted. After all, being a system programming language, these facilities are implementable in Peano and can therefore, if needed, be imported from an appropriate library - written in Peano!
- c. Programs written in the language should be easily understandable when read - i.e. readable! It was decided to take this decision because most of the problems found when correcting or enhancing a program occur because it is difficult to understand. This does mean that writeability has been sacrificed if a conflict was seen although this has not always been to the disadvantage of the 'typist'.
- d. The introduction of the sorely needed generic facilities must not lead to enormous complications within a compiler - or in runtime code! Three potential problems identified have therefore been deliberately 'legislated' away - different generic types are *never* pointer-compatible - generic routines are *never* compatible with any other definable routine type - generic modules must *always* be entire (not separated into a face and corresponding body). This last 'law' obviates the potential problem of iterative link/compilation or heavy runtime overhead code.
- e. Generally type compatibility should be structural except for such things as private types, where name compatibility would be used.

Once even a very preliminary design outline had been produced, it became obvious that this language could offer a great deal not previously readily available. It seemed to offer the possibility of defining everything from a complete operating system through to an advanced user environment with almost no need to use native machine code. It was decided therefore that Peano must be fully designed and implemented.

It is worth noting that the eventual design of Peano seems to be even more usable than originally suspected, since there is no need at all for the programmer ever to use machine code. All programming can be done in the very high-level Peano language - in what promises to be a very efficient way.

The decision to implement this second language meant that the original goals of this work had to be considerably revised. The team was faced with producing two compilers for two languages (Modula-2 and Peano), to run on two machines (VAX-11 and Z8000), under two operating systems (VMS (VAX only) and UNIX (both machines)). Because of this it was decided to look into a portable system design - hence the current project title.

### Source Language Portability

In order for the project as now conceived to be successful, portability has to be achievable at a number of levels, but particularly from the programmer's viewpoint, at the source language level. With languages which are aimed unashamedly at the system programmer, this means that even low-level constructs and ideas must be portable except, of course, where specific machine values are required in a program. This problem was tackled by looking at the differences and similarities between a wide range of machines which could be used. Apart from machine addressing problems, the most obvious one from the program writer's viewpoint is that of the exact definitions of predefined types! Nearly every make of microprocessor has different hardware arithmetic capabilities and consequently there are likely to be portability problems for quite ordinary programs.

Whatever the preferred sizes for arithmetic manipulation on processors of all kinds, there is frequently hardware support for a range of sizes or, in the worst case, software emulation facilities for multiple length arithmetic. In order to provide true portability where precision and/or numeric ranges are of significance to some program variables, it seems essential to introduce the idea of numeric type generators into the compiler system. A compiler is then at liberty to check that the requested size is available on the target installation and flag an error if it is not provided. This solution avoids the problem where a program appears to compile - but then produces the wrong answers!

Since there are three kinds of arithmetic in most languages, the system should provide for three numeric type generating functions

- a. INTTYPE - which generates a type on the objects of which exact signed arithmetic may be performed.
- b. CARDTYPE - which generates a type on the objects of which exact unsigned arithmetic may be performed.
- c. REALTYPE - which generates a type on the objects of which approximate signed arithmetic may be performed. The type generated is guaranteed to have at least one eighth of the bits as a binary exponent and at least three-quarters as mantissa.

### Modula-2 Language Extensions

The three type generating functions described above are provided directly in the Modula-2 language defined for this project. They may appear in a program at any point where a type definition may appear.

With these three functions and the formal notion of enumerated value types, it is now possible to define portable versions of all of the predefined language object types, definitions which apply to every machine implementation, only being affected by the constant Bits\_per\_Word found in the MACHINE library module. The definitions are

- a. CHAR = CARDTYPE(7) - where the character value encoding is in accordance with ITA No 5.
- b. BOOLEAN = (FALSE,TRUE) - note that this *can* be expressed as a single bit.
- c. INTEGER = INTTYPE(Max(Bits\_per\_Word,16)).
- d. CARDINAL = CARDTYPE(Max(Bits\_per\_Word,16)).
- e. REAL = REALTYPE(Max(Bits\_per\_Word,32)) such that Max\_Real is of the order of  $1e38$  and that at least six decimal digits of precision are available.

The implementation expects that all machine systems will provide real arithmetic to conform to the proposed IEEE standard (32, 64 and 80 (temporary) bits). If necessary, of course, the compiler can generate calls to appropriate emulation software routines to achieve this.

As already noted, the problem of expressing machine addresses must also be considered in attempting to



suspected, one in the consid- o), to both ect

provide complete portability. Fortunately most of this problem is syntactic rather than semantic. The definition of Modula-2 produced by its designer considers machine addresses as being indices in some completely homogeneous address space. Unfortunately many machines have several different address spaces and also several different, non-homogeneous, ways of viewing them.

The inhomogeneity of an address space is usually expressed as some page or segment number together with an offset into that page or segment. In order to express this form of addressing, it has become necessary to introduce the following modified syntax for addresses

address ::= [ segment-or-page-number '\*' ] offset

For all machines the absence of a segment-or-page-number implies segment or page zero. In the case of a machine with a homogeneous address space then any segment-or-page-number is ignored by the compiler system.

All the machines which provide different address spaces seem to differentiate between memory space (which may, say, be segmented) and input/output address spaces of different kinds - often requiring different instructions to access them. This separation of input/output from memory seems to be such a general idea that it has been decided to introduce an additional type generator. If this were not done then it would not be possible for the compiler system to detect the difference between the declarations

Interrupt\_Vector [0 | 2] : WORD

and

Wide\_Channel [0 | 2] : IOWORD

both of which could be valid in the same program! The inclusion of the type constructor PORT, however, enables the programmer to define

IOWORD = PORT FOR WORD

which easily provides the necessary differentiation in a neat and unambiguous manner. Both of the declarations are then valid and the compiler can generate appropriate code for input and output - using the PUT and GET routines introduced into the SYSTEM pseudo-module (see below).

The only additional extension considered for Modula-2 concerns the need in many machines for bit group manipulation, particularly in "special registers" or memory locations. These locations are often 8, 16 or 32-bit wide, consisting of a number of fields, some Boolean and some numeric. It seemed that the ability to separately manipulate each field would make programs considerably more readable and also enable the compiling system to take advantage of any special bit extraction/insertion instructions which are now available on many machines - for efficiency! The only way to do this consistently is to introduce the idea of a field offset in a record. By adding in brackets after the field identifier a bit offset from the beginning of a record it is possible to build up a packed record. Since there are occasionally gaps in such records, it has been decided that overlapping is NOT permissible except by using the variant record facility, but that gaps are permitted.

### The SYSTEM Pseudo-Module

As in the original design of Modula-2, the majority of hardware dependencies are confined to the pseudo-module SYSTEM which is provided for the programmer, but known to the compiler since special code is generated to access these very low-level machine features. The facilities required in this pseudo-module can be considered in three separate groups

- a. Those facilities defined as part of the original Modula-2 language specification - which are mandatory.
- b. Those facilities required to access low-level machine features in a syntactically and functionally portable manner.
- c. Those facilities required to port the compiler system between different machines.

It is important to note that the requirements for low-level access and system portability apply generally for any language or machine covered by this project. To this end, for example, it is necessary to provide primitives to support multi-processing in addition to the simple Modula-2 coroutine mechanism. Facilities to handle interrupts (whether hardware or software) are also needed. Together these two requirements mean that the implementation of the standard procedure TRANSFER must be made visible to the SYSTEM user if he wishes to make direct use of them. This "opening-up" of TRANSFER has the added advantage that it allows the provision of a machine-independent hardware and software interrupt mechanism - providing that a TEST AND SET

6

(1) ADR -  
(2) SIZE  
(3) TS  
(4) 'C

facility is added. This results in the need for three uninterruptible operations altogether - SAVE\_CONTEXT, LOAD\_CONTEXT and TEST AND SET. Coroutines, interrupts, multi-processing and multi-programming can all be implemented with the assistance of these three primitives.

The second important problem which can be solved by the redefinition of SYSTEM facilities is that of object sizing. Although the numeric type generators work for numeric objects, the idea of a memory unit is embodied entirely within SYSTEM. It is necessary therefore to include a universal definition of 'word' and 'address'. This is, unfortunately, not possible directly. Many computers, for instance, have a variety of sizes of object which can be addressed directly and there is often a difference between any of these and a machine address size! In order to specify these ideas portably it was decided to categorise machine objects into two groups

- a. Storage objects which may be of two sizes, the preferred machine addressing unit, called WORD and the smallest normally addressable memory element, called BYTE. These may also be referred to as major and minor addressing units respectively. Where machines embody several sizes of addressing unit all known machines provide multiples of a minor addressing unit - which may therefore be expressed as an array of bytes.
- b. Addresses of machine locations, either as an ADDRESS in normal memory or as the address of some port in input/output space or, finally as a register. Since registers are always intimately connected with the instruction set of any machine which has them, it was decided to access them solely through special routines. The problem of machine address arithmetic raises a further minor portability problem for machines with paged or segmented addresses. There must be a way of carrying out such arithmetic portably. After some indecision, because of possible sizing problems, it was finally agreed that the *offset* of an ADDRESS object would always be compatible with the standard type CARDINAL - irrespective of sizing differences which would be hidden by the compiler if necessary. Out-of-segment results for such arithmetic are the programmer's responsibility.

The removal of machine registers from the type domain to the functional one has a further advantage for the compiler. The retrieval or setting of register values, at least for general registers, inevitably upsets the compiler register allocation mechanism - which could avoid such troubles by clever optimisation in many cases.

The problem of program portability does not arise in the case of register usage, since this is inevitably very machine dependent! Rather, the problem is one of programmer portability! A large number of machine designers have adopted as wide a range of mnemonics for registers at Assembly Language level as there are machines. In order to provide some measure of standardisation, registers are identified in the runtime library module MACHINE as values of an enumerated type Reg\_Kind. The values R0, R1, R2, etc are designed to correspond directly to 0, 1, 2, etc to ensure portability of existing programs. These names also correspond, for example, to the letters A, B, C, etc used to identify general registers on some microcomputers. Where appropriate the names SP (Stack Pointer), PC (Program Counter), Status\_Reg and Flags\_Reg also have standard meanings. All other values unfortunately have to be machine-dependent.

The pseudo-module SYSTEM therefore contains a group of type definitions and some procedures and functions. The list which follows serves merely to identify those facilities offered

- a. Pre-defined types
- (1) BYTE - minor addressing unit.
  - (2) WORD - major addressing unit.
  - (3) ADDRESS - in main memory.
- b. Pre-defined procedures
- (1) SET\_REGISTER - alter a specific register value.
  - (2) SAVE\_CONTEXT - save the processing context of the calling environment.
  - (3) LOAD\_CONTEXT - restore the processor context to be that of the indicated process.
  - (4) TRANSFER - a combination of saving and loading contexts.
  - (5) PUT - data to an output port.
  - (6) GET - data from an input port.
- c. Pre-defined functions which return the values



- (1) ADR - address of an object.
- (2) SIZE - size of an object in bytes.
- (3) TSIZE - size of a type of object in bytes.
- (4) REGISTER - contents of indicated machine register.
- (5) NEWPROCESS - a new process (as the address of a Process Control Block).
- (6) TEST\_AND\_SET - the boolean flag value.

The detailed specification of all these facilities will be found in the Modula-2 Language Reference Manual. The facilities are also directly available in Peano if desired.

### Runtime Environment

Before any code could be written, it was realised that the eventual portable runtime system had to be designed - and implemented - before almost anything else in the project could be coded! This essential need for portability of runtime environment meant that all actions must be abstracted from the operating system. In fact, it was thought desirable to implement most of the facilities as abstract data types, although such things as number-to-string conversion, heap storage allocation, etc are not appropriate to this type of treatment. The essential point in the design is that there must be a layer of at least one module thick between the native operating system and the user program - even if this layer is merely a definition of the operating system facility where this proves to be identical to the required semantics.

The abstract data types offered are fairly conventional in most respects - FILES, STREAMS, RANDOM FILES, TERMINALS, EVENTS (for synchronisation) and PROCESSES. There are, however, three which merit more detailed discussion in relation to design for portability

- a. NAME. This abstract data type represents the notion of the operating system manner of representing the identity of an external (e.g. file) object. The way in which this identity is coded is, of course, quite different from the way in which a user may be required to express it as a string to his command language. These discrepancies have led to the notion that there are five logical components which *may* form a means of identifying an external object. These are Node (in a network, say), Directory (the actual lookup list for the object location), User-chosen string - the conventional idea of a name string, Kind - the kind of information represented in or by the object - which may have some significance for a local operating system and, finally, the idea of a Version or Generation of the object. Not all file systems support all of these, but no file system supports more than these so far as is known - even with multiple levels of directory. Using these strings (or a cardinal number for Generation) operating system specific implementation routines can hide the command language syntactic detail and the OS internal form from the programmer - UNLESS literal strings are built-in to the program! Even then these could be secluded in a little installation-dependent module - as, in fact, has had to be done with a few facilities in the compiler system itself.
- b. Arguments - *almost* an abstract data type! This facility is designed to allow a user calling a program to pass it arguments of almost any reasonable kind which can be expressed as strings - as part of a 'command line'. While the arguments themselves consist of a separating preceding character and a string of characters, the method of implementing this facility varies widely from system to system and may, in some systems NOT be available. It is for this reason that the argument facility is not considered to be an abstract data type proper - it cannot ALWAYS be implemented. However, it was decided that an interactive 'read arguments' facility should also be provided in order that programs could rely upon being able to obtain arguments from the invoker.
- c. EXCEPTION. Modern thinking about programming seems to have concluded that the idea of an exceptional occurrence, which could not have been foreseen in the design of a program, which prevents some procedure from completing its actions correctly, but which may be recoverable at some other point in the program, is a sound component of a good programming language. The data abstraction facility offered by Modula-2 suggested that it should be possible to offer exceptions in the style of Ada exceptions without a great deal of difficulty. The preliminary version of this idea did not encompass propagating exceptions from child to parent process, and therefore seemed straightforward to implement, needing only a knowledge of how the compiler built its run-time stack. Unfortunately, a little deeper thought revealed that all was not quite so simple! This point is discussed in detail below.

The design of the runtime facilities relies upon a number of machine-independent and OS-independent definition modules, which, together with a very few OS-dependent definition modules can be implemented in

7

different ways for different Operating Systems. The need for OS-dependent definitions occurs only at the lowest level of the library, where data structures needed in other library implementation modules, but nowhere else, are peculiar to one system. Similarly, although the routines offered by an operating system may differ in their detailed semantics, parameters, etc, the lowest *Opsys* module produces the calls required by the higher library levels. These then adapt the OS semantics to the required portable definitions.

Having adopted this type of philosophy has meant that the fundamental lowest level either describes machine/OS structures and constants or offers syntactic call commonality across systems. The next higher layer is a service layer which produces standard portable semantics for routines/functions of fixed name, either for direct use by the programmer or, more likely, for use in implementing the higher level functions which programmers expect. For example, *Read* in *Opsys* becomes *Byte\_Read* in *Byte\_Channel* (raw serial input) and *Block\_Read* in *Block\_Channel*. These are in turn used differently in the *TERMINAL*, *STREAM* and *RANDOM\_FILE* abstract data types - although they may, of course, be used directly.

In addition to the 'almost-abstract' Arguments facility, any clock or timing facility offered in the standard library *Clock Timers* module can only be used in an installation where there is a hardware clock from which values may be derived.

Since this runtime library is being designed not only for Modula-2, but also for Peano, Pascal or any other language which is implemented, it has to be designed to run in a very wide range of environments. One of these possible environments is the multi-processor environment - whether these are in the same box or many kilometres apart! The realisation that this was necessary led to a complete reappraisal of the *PROCESS*, *EVENT*, *EXCEPTION* section of the library. No longer could it be assumed that interprocess communication was all in one machine, let alone all in one memory area - the original starting point for the design. Furthermore, it was soon realised that, within one process an exception was completely synchronous (even if not expected). When propagated to a parent in an attempt to find a handler such an exception becomes asynchronous. The need for a facility to generate and react to asynchronous software interrupts was recognised. While this is not necessarily connected with hardware interrupt mechanisms, it does rely upon there being at least one uninterruptible instruction within the processor or at least a facility to bar all interrupts and re-enable them. Since Modula-2 has the *SYSTEM* pseudo-module, this is a convenient place to introduce the programmers' requirement for uninterruptibility in a machine-independent manner. The function *TEST\_AND\_SET* was therefore added to *SYSTEM* as described earlier.

The library therefore consists of some twenty-six modules which provide the kinds of facilities normally associated with a modern high-level language programming environment. These facilities are fully described in the User's Guide and the Reference Manual for the Runtime Library.

#### System Dependencies

The general feeling that the original project plan was becoming more and more involved as design progressed was never far from our thoughts, but the needs still clearly existed and the problems still needed solution. Now that a first design was beginning to take form, it was pertinent to look at the implementation problems which could arise and try to ensure that neither one (design or implementation) made the other more complicated!

In any attempt at producing portable software there are a number of relatively straightforward aspects to be considered. In a portable compilation system these problems are compounded by the need to consider

- a. The source language being compiled.
- b. The host machine on which the compiler is running.
- c. The host operating system under which the compiler is running.
- d. The target machine for which code is to be generated.
- e. The target operating system under which the generated code will be run.
- f. Any interactive device involved in generating/debugging/testing programs.

**Source Language** The problem of source language dependencies is central to the whole project - at least in the sense that the compiler code itself is supposed to be portable. However, the whole project is also concerned with source language dependency in making sure that it is viable for a non-trivial selection of languages which may be needed on a variety of machines. While no detailed study has been conducted, a preliminary investigation shows that most well-known languages - with the possible exceptions of PL/I and Ada - are expressible in a



common intermediate language (see below). The most important aspect of this dependency therefore is to minimise the amount of compiler code which needs writing/rewriting when implementing a new language. In particular no compiler house-keeping or data structure manipulation should need revision!

**Host Machine** Where host and target machine are the same, as in many practical cases, then no specific host-machine problems arise. Unfortunately, with an ostensibly portable system the overall design must prevent repercussions of a change of target machine from spreading throughout the entire compiler. The main problem is the one of literal conversion, where source character stream is converted into a host-related bit pattern only to need later conversion from this into a different target-related pattern in output code. Loss of accuracy or rounding errors could occur unavoidably in any numeric conversion involved. The compiler must therefore only make a conversion from source literal when it is known whether or not a value is needed at compile-time, run-time or, very occasionally, both!

**Host Operating System** Although the host operating system will have no effect at all on the resulting compiled code in a well-written compiler, it does have a very considerable effect on whether, when ported to another machine and operating system the compiler will even work at all! There must, therefore, be a completely portable run-time environment.

**Target Machine** One of the main purposes of a portable compilation system is that it should be readily adaptable to different target machines. To do this sensibly, however, implies that target machine dependencies should be restricted to as small a portion of the compiler as possible.

**Target Operating System** The operating system under which the generated code is to run, if indeed there is one at all, has a profound influence on a portable compiler system. Firstly, it has to enable programs to both use target operating system facilities and also to themselves become new operating system facilities. This requires that the compiler be able to generate code both for its own portable calling convention and also for the target operating system calling convention - as may be required - in the same program. Secondly, the portable system must be able to interface with the target operating system linker or loader (again, if any!). This essentially requires a separate portable linker which will put together modules from many compilations into a correct single program form for operating system native linking/loading.

**Device Dependence** In most compilation systems there is relatively little need to pay attention to the terminal devices which programmers may be using to develop their programs. In a portable system of the kind being now envisaged, however, it seems fairly important to provide, at least optionally, a debugging facility - which, naturally, must also be portable! While it was agreed that debugging was to be a secondary design consideration, it was perceived that debugging is only a particular form of editing. This in turn led to general editing considerations and, consequently, to the need to consider device independent editing. This topic will be discussed in more detail in a later section.

With six quite different dependencies to consider, it was immediately apparent that implementation would be far easier - and safer - if a potentially restrictive design rule were to be adopted

*The implementation of a module shall never depend upon more than one of source language, host machine, host operating system, target machine, target operating system or interactive device.*

The adoption of this rule has certainly ensured that some import/export lists are a little longer. It has meant even that extra code has had to be written in some modules. Above all, however, it has made all modules easier to write and understand - a very important point where several hundred modules are involved!

#### Intermediate Code Selection

The key to a successful design of this nature was soon seen to be the choice of the main machine-independent intermediate code. In view of the need to express the semantics of two widely differing languages which were ostensibly both suitable for writing system software, it was decided to produce a set of requirements and attempt to select an optimum from existing widely used intermediate codes. This would inevitably have the advantage that machine code generators would already be available, saving considerably on the total effort required.

The requirements which had to be satisfied by the intermediate code were originally specified as having the ability to

- a. Express all conventional high-level language constructs (loops, alternation, routines, functions, etc).
- b. Express references to objects of other modules which were to form part of the same program.
- c. Express references to *foreign* objects in a machine independent manner. These were generally thought to

be operating system variables/routines.

d. Contain the structure of the original program for optimisation and debugging purposes. Since both Peano and Modula-2 are suitable for writing system software, it was considered that the compiler system must provide the highest possible degree of optimisation.

e. Contain the semantic actions of a high-level program absolutely independently of the target machine code, target machine architecture, target machine operating system (if any) and source language (within a wide class).

Since Modula-2 is a Pascal related language, the natural starting point in the investigation was to look at P-code and its variants, including Q-code - a partially attributed form. None of these proved suitable since they do not contain inter-module reference facilities nor the wider typing information needed by system software which has to carry out multiple length and kind operations (e.g. byte unsigned arithmetic, 11-bit signed arithmetic or even 3-bit modulo arithmetic). This structure of the original source program has been reduced to linear form in any of these code variants, making the availability of certain permissible optimisations much more difficult to detect. A look at other well-known intermediate codes - including Z-code, O-code, etc - showed that none were suitable to the required task without considerable modification.

The need for potentially radical alteration suggested that it would be far better in the long term to design a completely new intermediate code. Using the well-known stack model as certainly being machine independent a preliminary version of such a code was designed to satisfy the given initial requirements. After a great deal of hard design work, however, it became apparent that, for Peano at least, the symbol table information required by an importing module needed to be able to contain intermediate code!

The form in which symbol information was to be passed between face/definition and body/implementation had, at that time, not been considered further other than to make the preliminary decision that it should not need to be parsed again when read from file - any work only involving simple conversion from file-linear to structured form using some structure coding in the symbol file. In common with most modern compilation systems the unexpressed intention had been to make the symbol table fundamentally *tree-form*. The expression of this structure in a symbol file and the potential need for this *tree-form* to contain intermediate code now led to the realisation that this intermediate code itself should also be *tree-form*.

After considerable further development of these ideas in discussion, together with a study of Diana - the Ada intermediate language - it was decided that Diana was too specifically Ada-oriented and at a rather higher level than needed by the class of languages being considered in this project. Those other languages which had been thought of included Algol-68 at the more complex end of the spectrum and C at the opposite end. The ideas born during the design of the stack-based preliminary code were therefore reviewed to see what would be useful to carry over into the design of a tree-based code. Apart from a number of code kinds which would need structural modification the most important feature to be carried over, which effectively sets the limit of the intermediate code, is the need to parameterise the code for storage sizes. This is needed so that offsets and addresses can be handled correctly without enormous complication. This is one of the major differences from Diana - the intermediate code was to contain target machine object sizing. This had to be a parameter of the code rather than a design feature so that code could be generated for objects being held on everything from a conventional 8/16/32-bit kind of hardware to a less conventional 23-bit word or 11-bit byte, say!

So Rcode was born. There are eight fundamental classes of code in this intermediate code (which is fully described in the Rcode Reference Manual) which offer the ability to use it as a symbol file code, as an interpass code on a multipass compiler and even as a fully interpretable code - when a machine code generator is not wanted - or available! These classes are :-

- a. Group 0 - Pseudo-operations - indicating Rcode manipulation rather than source language representation.
- b. Group 1 - Declarations - of areas and objects - in terms of size only.
- c. Group 2 - Values - ways of expressing values of all kinds.
- d. Group 3 - Basic Operand Manipulation - moving objects of all sizes.
- e. Group 4 - Control - calls, loops, alternation, sequencing, setting, jumps, etc.
- f. Group 5 - Arithmetic - of many kinds on objects of many sizes.
- g. Group 6 - Machine/System dependent extensions - primarily permitting access to 'special' operations - but including I/O, register handling as a standard.



- h. Group 7 - Type - definition of source language type information for symbol table, debugging, etc.

Each class of code contains a mixture of elements which are terminal and non-terminal nodes of a tree structure which can embrace an entire compilation unit including its imported units if desired. By arranging to generate Rcode in a pre-order form it is possible to file it easily and even process a tree before the tree is completely built - if desired.

### Compiler Architecture

The choice of Rcode as the intermediate code has had two major impacts on the overall design of the compilation system. The first effect noticed was that the front and back ends (in the traditional sense) of the compiler may be completely independent of each other. In turn, this independence means that implementation for a new hardware machine or the implementation of a new high-level language only involve one end or the other of the compiler.

The second major impact of Rcode was on the choice of compiler phases and on the solution of the associated small-machine problem. The original Modula-2 compiler for the PDP-11 had solved the small machine problem by not only having a number of passes and storing interpass data on temporary files, but also by treating compilation as a sequence of virtually independent processes which could be explicitly loaded and run one after another. Other Modula-2 implementations seen, retain data files of interphase information but allow operating system memory management handling to look after code space. These models suggested that a truly portable system must allow for at least two and preferably three potential architectures into which all the code can be fitted :-

- a. A simple one pass sequential-phase compiler in which all code would be available at once and the total non-symbol-table data space would be kept within reasonable bounds. The symbol table size is, of course, programmer module dependent and could therefore need to be limited in some machines, at least in principle.
- b. A minimum space multi-pass sequential phase overlaid compiler for small machines - similar in principle to the original PDP-11 Modula-2 compiler.
- c. A constraint free one pass concurrent-phase compiler for either large machines or multiple-processor systems. All phases can execute concurrently, limited only by the availability of "input" data for transformation by each phase.

Irrespective of the choice of architecture for a particular system, the phases had to be chosen to minimise interphase overheads. Essentially this means that phases must be chosen to be self-contained in a way such that the only difference between the three architectures described is the way in which the next data-item needed is provided (and the last one disposed of) - from memory, from file or from memory with waiting. As far as possible this should mean that the smallest possible amount of data should need to be retained by the compiler between phases.

The main items needed in more than one phase are identifier, literal and symbol tables. It is necessary therefore that these can be simply filed and reconstructed if needed. Identifier and literal tables are merely character arrays and cause no problem. The symbol table is, unfortunately, much more complicated. Fortunately Rcode comes to the rescue in the way already indicated. The only complication is in the conversion from file-linear form to the highly structured form in memory. Although such a conversion may be a time overhead in a small machine compiler, it is not a space overhead since the relevant code is needed functionally when reading symbol tables from the files of imported modules!

The phases of the compilation process could now be clearly identified, at least in the compiler front end, as

- a. Lexical analysis producing an identifier table, a literal table and a symbol stream.
- b. Syntax analysis producing a syntax tree of symbols and a symbol table as well as retaining identifier and literal information.
- c. Semantic analysis, producing an Rcode tree and retaining literal or identifier information only as needed in this tree. Symbol table information is retained only insofar as is needed in Rcode form in the 'output' from this phase.

In addition to these functional phases which are common for both definition and implementation modules, the front end necessarily contains elements of listing, cross-reference production and, of course, error notification functions.

While there was obviously going to be at least one phase to the back end of the compiler, a number of other problems had to be solved before this could be investigated more deeply.

### Compiler Syntax Analysis

While the existing Modula-2 compilers available use a form of syntax-directed top down analysis inherited directly from their Pascal progenitors, this form of analysis is highly language dependent, almost on a line-by-line basis. Although there is always considerable attraction in not having to 'reinvent the wheel', the thought that this type of analyser was probably the least reusable of all for different languages, led to its rejection. The kind of analyser needed for a multi-language compilation system should, it was considered, be assembled in building block fashion from selected off-the-shelf components.

Over a period of some two years up to 1983, preliminary work had been carried out within the Department of Computer Science on building a syntax-directed editor for the Pascal language. Independently, a long-term review of the whole subject of editing had also been carried out; undoubtedly this proved that editing is one of the most emotive subjects on which any hundred users will have at least a hundred and one different, strongly held opinions! This study soon showed, however, that editor syntax 'analysis' problems were indeed very closely related to compiler syntax analysis.

A decision was therefore made to use common code for analysis in both editing and compiling!

Since one of the features required by an editor during syntax analysis is the ability to independently parse almost any possible program segment, it would be essential to adopt a syntax analysis technique which would be very general. Any such "intelligent" editing facility needs to be able to cope with not only programming languages, but also everything from plain text to binary files, via graphic images and text formatting languages! This wide variety of data structures could mean that a truly generic editor was required - indeed a portable generic editor. The details of the development of the editing facilities in the project are discussed in a later section. The syntax analyser, being common between editor and compiler still had to be designed.

Once the decision to incorporate intelligent portable editing into the project had been finally agreed, the obvious answer to syntactic analysis was to provide a table-driven parser since tables can be readily loaded or changed dynamically during user interaction.

With the potential involvement of many languages (some formal, others less so), it was obviously going to be almost essential to make use of an automatic table generator. Since the UNIX<sup>†</sup> tool yacc was available it was decided to make use of this as the basis around which to develop an automatic table generation system for the compiler/editor syntax analyser. The disadvantages of yacc for the project were that

- a. It produces the appropriate tables (and an analyser) in the C programming language.
- b. The automatic error handling provided within the analyser is very crude and completely unacceptable for a student production system.
- c. The token system of yacc derives from the use of its companion tool lex. This is of little use where a more structured approach to tokens is required.

These three factors have meant the adoption of a special purpose approach to the use of yacc in generating a syntax analyser.

**Micro-Syntax** The subject of lexical analysis (or micro-syntax) could also have been treated by a general purpose production tool like lex, but it was considered that a hand-built lexical analyser would be far more efficient. In addition to efficiency considerations, the tokens must be producible from a variety of sources - changeable dynamically at will

- a. From a text form source file - like a conventional program source!
- b. From some internal editor string buffer - again in text form.
- c. From a linearised parse tree file - saved by the editor from last time.
- d. From a parse tree being manipulated by the editor.

In addition to the impracticality of using anything but a handwritten token producer to cope with these dynamic changes, the close integration of editor and compiler also brought a complication not originally envisaged. The conventional compiler lexical analyser reads and discards comments. The editor, when editing the text form of

---

<sup>†</sup>UNIX is a trademark of AT&T Bell Laboratories in the USA and other countries.



source sees the comments and can display them for the user - when editing a saved program in tree form these comment strings will have normally been removed by the compiler. To obviate this problem, a comment-token has had to be introduced - which a compiler ignores (except when generating it) but which an editor uses to find the string on some comment storage file. A number of other non-conventional tokens have also been introduced to enable the editor to direct compilation in its own way.

**Error Handling:** Once again the conflicting requirements of editing and "straight" compilation for the handling of syntax errors in different ways requires special purpose routines. If necessary, for example, an editor may wish to stop analysis on the fly to allow correction of errors before continuing if appropriate. Since incomplete or incorrect error recovery is a major weakness in many current compilers, making them unsuitable for student use, it was decided that an attempt would be made to produce high-quality error recovery code/tables automatically from a specification of syntactic recovery points and recovery sets. This very important aspect of the project is still the subject of further development although the fundamental algorithms required have been proven; all that remains is to investigate improved table compaction and the generation of faster recovery code.

**C to Modula-2 Conversion** The parser tables produced by yacc are in the form of initialised C arrays. The associated code generated is fixed independent of language, which means that a hand-coded replacement may be readily written in Modula-2. This rewritten version does, of course, take into account the complicating factors of different sources and the editor's needs. These have meant that, in addition to simple recoding in the different language, the following new features have had to be incorporated

- a. The conversion of the semantic actions on reduction into an array of procedures indexed on the relevant production rule. This mechanism enables different compiler architectures to be implemented by, say, replacing all routines by one file-writing routine to produce an interpass file. This same procedure array may alternatively be loaded by the editor with its own action routines as and when required.
- b. The normally fixed start symbol can be set dynamically so that a partial routine, statement or even expression can be analysed instead of an entire compilation unit - as the editor may require.
- c. The same array of procedures technique can be used to provide different error recovery actions dependent upon the error recovery tables. The details of these routines are not yet final.

**Table Production** The production of hand-written routines for syntax analysis is, of course, the minor part of producing a table driven analyser. The production of the tables themselves is at the present carried out in three stages

- a. A pre-processor takes as input a form of syntax description which includes an indication of recovery points and permissible follow sets by means of an angle-bracket notation added to the BNF-like specifications used for input to yacc. The output from this preprocessor is in two parts - one is a standard version of the syntax as expected by yacc, together with an error recovery data file.
- b. yacc itself, taking the preprocessed syntax as input and producing its standard form of output file.
- c. A post-processor which takes the yacc output and the error data file as input. This produces four Modula-2 modules, the definition and implementation modules of symbol definitions (lexical, error and non-terminal values as enumerated types) together with the corresponding modules of syntax tables including the initialisation code.

This table production mechanism could be refined by rewriting yacc to produce Modula-2 output directly to make the entire system, together with its production tools, portable. This will eventually become essential so that the system can become self-extending as required. This would then enable the editor to create new structures for editing and then edit them by loading tables and routines dynamically.

**Compiler Control Syntax** The occasion afforded by the use of yacc to rewrite the syntax of Modula-2 in the correct form, led to the realisation that one of the needs of a compilation system which was suitable for producing system software was the ability to provide special control over certain segments of a source module, for example to indicate special alignment of objects or to inhibit optimisation, etc. Since the kind of compiler control being considered is closely related to language structures, it seemed sensible to build the compiler control pragmas into the language syntax - as options, naturally.

#### Compiler Directives

The control of compilation options may be effected by directives which may take the form of arguments to the compiler call or as pragmas buried in the body of the program source text (tree). Some options were

originally conceived as possibly being provided in both forms. However, further consideration showed that the options needed were either global - over an entire compilation - or related to some specific syntactic structure. For these reasons, the argument directives and pragma directives have been chosen to be mutually exclusive, the former being in fact the class of directive commonly found in many more conventional compiler systems. The majority of directives provided are independent of the source language.

**Common Arguments** In addition to the identity of the source file, which will be expected to be either tree or text form, the optional arguments which may be passed to a compiler have been agreed to be

**a. Directory =directory-file-name**

This argument and its parameter instruct the compiler (and linker) where to find all imported modules required for the current compilation (in addition to those which may be obtained from the standard runtime library), whether symbol or code files. The directory file is expected to be a simple text file containing on each line, first the module name and then the system-dependent names of the files containing the symbol and object forms of the module - on the same line.

**b. Query**

This argument forces the compiler to interactively request the user to enter the system-dependent names of the files containing the appropriate form of a module needed by the compiler. If no file can be found then compilation will be aborted.

**c. Output =object-file-name**

This may be used to override an installation dependent default object file name.

**d. Symbol =symbol-file-name**

This argument may be used to override an installation dependent default name for a symbol file.

**e. List [=list-file-name]**

In addition to directing the compiler to produce a listing, the addition of the optional list file name overrides the default installation dependent name which is provided.

**f. CrossRef [=crossref-file-name]**

This argument directs the compiler to produce a cross-reference listing of the compilation unit on either the named file or, if no file is named, then on the listing file (whether or not a listing is also being produced).

**g. MachineCode [=code-file-name]**

In addition to producing code for the linker (which will always be produced in the absence of errors), this argument requires the compiler to produce a human-readable form, either - by default - on the listing file or on the named file if the parameter is present.

**h. Debug**

This argument option directs the compiler to pass symbol table information through the machine code generator for use by runtime debugging facilities. Although the directive was originally introduced as a 'hook' for a later project phase, it was realised quite early that portable debugging must also be provided! This is essential if the project aims of true portability are to be achieved. This meant, in turn, that the editing view of debugging has had to become a central feature of the design. This aspect of the project is treated in detail in the section on the Editing Environment.

**i. Variant =variant-name**

Several processors are made or operate in variant forms, where the differences are so small as not to warrant a completely different compiler - or even merely a change of code generator. Typical examples are the Intel 8086/8088 microprocessors which have almost identical machine code (even Z80/8080 processors) or the Zilog Z8000 series processors which are either made or can operate in segmented or non-segmented addressing modes. The variant names chosen will, naturally be machine-dependent, but one of the options will always be the default.

**j. AddressCheck and RangeCheck**

These two arguments control the production of runtime checking and reporting code. For some languages one or both will be the normal default (e.g. Pascal) whereas for others checking code will not normally be generated.

**Pragmas** Most of the pragmas offered as standard for any language are designed to affect the way in which machine code is generated without affecting the meaning of the program. The exceptions to this are noted in the following subparagraphs which describe the different classes of pragma



a. **Pseudo-Syntax.** This class of pragma modifies the language syntax to enable a program to either import or export objects from/to the native operating system. The use of the External pragma (with a string parameter) adjacent to the identifier of a declared object instructs the compiler that this object (with the given external name, string) will be linked into the program by the native operating system linker. This may of course apply to any runtime object. The Global pragma is the converse, specifying that the object in the module will (eventually?) become part of some native operating system library. Whichever of these two pragmas is applied to an object it instructs the compiler to adopt that target operating system conventions when generating code for handling these objects.

b. **Type-Definition.** The five pragmas in this class (Align, BitPos, BitSize, ByteSize, MachineType) are aimed at specifying, where necessary, the way in which storage for objects of the associated type is to be used or aligned. These can be used to ensure that particular machine-specific sizes, instructions, etc will be generated by the compiler.

c. **Constant\_Definition.** One of the awkward things about high-level programming languages is that constants are abstract notions which have to be given a representation in computer storage in a practical program. The reality of this can be used "illegally" (in most languages) by allowing the compiler to generate pointers to such run-time read-only storage where it is desired to minimise total code or data size. The Referrable pragma may be included in any constant definition to permit this form of minimisation. Care must be taken that read-only use is made of such pointers.

d. **Code-Related.** As the name implies, this group of pragmas is directly intended to influence the way in which code is generated for some particular structure. NoOptimise suspends all code optimisation allowing code generation which may be required to produce a side-effect not valid in the language concerned. Unsafe when applied to a type suspends all type checking in relation to objects of that type. The other pragmas in this class are NoImmediate (do not evaluate at compile time), NoCheck (suspend generation of checking code), Inline (allow compiler to incorporate inline if cheaper) and Macro (instruct compiler to treat as a macro and use call by name argument semantics).

**Language-Dependent Control** No language dependent pragmas have been identified for Modula-2 or Pascal. Peano, primarily because of its two-level generic nature has four - NoPrivate, NoUnconstrained, PlainCode and Rcode - which are fully described in the Peano Language Reference Manual.

### The Editing Environment

As already indicated in earlier sections, portable editing, debugging and compiling facilities are all evidently needed if the original aim of moving students (almost) transparently from one machine to another was to be achieved. While it was not the original intention to provide such a completely portable facility, it was realised that the best way to move a user to a 'new' machine is not to move him! In other words (s)he should not need to be aware that a different machine is being used. For the purposes of the project therefore it was decided that portable editing, compiling, debugging and program running facilities were the minimum necessary. These facilities should be invocable in a portable way!

Rather than ask a student to interact with a wide range of new tools, it became evident that the minimum disturbance from his previously learnt user environment was desirable. Having used either or both of VAX/VMS or UNIX operating systems and any one of a number of editors and debuggers the choice for the design team was not easy. As editor design progressed, however, it slowly became obvious that a generic editor would necessarily offer some quite different features to other editors used. This would therefore be one quite important change for students. It was decided to minimise further change by incorporating debugging (only a glorified editing function) symbol tables into the data structures known to the editor and to add, therefore, a *run* command, enabling the program being debugged to be run/stepped, etc.

The final major step taken was to realise that the editor was now a complete user environment - since it is entirely up to the user interactively to decide whether or not to use the *debug* option to the *run* command. The project had suddenly become the parent of (in UNIX terms) a shell!

The original concept of a generic editor had arisen from an apprehension that the development of intelligent editors of all kinds would proliferate yet again the multiplicity of languages to be learnt by all but the very simple computer user. Even before completion of the survey of the enormous variety of editors, both commercial and experimental, which are available, it was realised that if it were at all possible, then a generic system would be ideal for a portable system. It was not until some outline design work was undertaken, however, that the real impact of this facet of the project became apparent.

15  
16  
The whole oppo sys Th n

In reality an intelligent editor is seen to be far more than merely a modifier of data - a fairly trivial task in itself. The primary functions of an editor are to provide a number of mapping facilities

- a. The most important mapping of all is between the user's input and the actions of the editor itself. In essence this is, crudely expressed, a command interpreter. Without the availability of this mapping then nothing else could be done!
- b. Mapping in the converse direction to provide output for the user, promoting useful interaction, is also required to present the editor's comments, queries and confirmation messages to the user.
- c. In similar vein, the user of the editor will almost certainly require the ability to find out what various portions of the object being edited contain. The editor must therefore be able to map its internal form of this for user output.
- d. To make any modifications other than deletion to the object being edited, the editor must also be able to map user input into its internal representation of the object.
- e. Yet another pair of mappings is required between the form of representation of the edited objects which the editor may use internally and the form in which the objects may be permanently storable - between editing sessions!
- f. Finally, a portable editor must be independent of the underlying machine and hence be able to interact transparently with its operating system - mapping the semantics into a portable version. This particular mapping is naturally provided by the portable runtime library system already referred to.

**Storage Mappings** Of the three remaining pairs of mappings, the way in which an editor reads and writes its permanent storage versions is almost entirely within the province of the editor designer so long as the methods adopted do not depend upon the object size or structure and so long as there is a mapping provided which will produce a version of the object which can be understood by other using programs! In essence this is merely a matter of careful design with data parameters which are changeable dynamically.

**User Interface Mappings** The editor must ideally be able to provide a uniform interface to the user, whether in respect of command and response mapping or object data mapping. Both of these pairs of mappings therefore have one very important part in common - the device-dependent aspects. Considerations of portability in respect of interactive devices may be looked at in a variety of ways. The historically oldest approach was to use a Lowest Common Denominator of device functionality to ensure that identical usage of 'keyboard' and 'printer' or 'display' resulted. This is most unsatisfactory for a generic intelligent editor.

Consider the need for two users of one system to use the same editor from different terminals - one a glorified teletype, the other a sophisticated graphic work station. Both users need the same functionality, but the teletype user must use different input representations for a command than those available to the workstation user (say, menu and mouse facilities). The first major editor design decision had been made - logical commands required for intelligent editing would be chosen entirely independently of the interactive devices which could be used.

A corollary to this decision was that interaction necessitated some form of non-printing output to the user, whether visual or audio seems irrelevant, just as the use of audio, keyed or pick-device seems irrelevant for input.

A further corollary of this decision was that, at least theoretically, it should be possible to make the editor human language/script independent by associating any further device-user translation with the device mapping software. This aspect of the project will not be implemented for some time although the early design decision is enabling appropriate hooks to be incorporated into the first production software.

**Command/Response Mapping** Once the device-dependent aspects of handling user commands and responses has been carefully abstracted into separate code, the rest is symbol manipulation and interpretation. While the major step of using the compiler lexical/syntax analysis code to carry out the syntax analysis of editor commands has not yet been attempted, there is little doubt that this will be done before project completion. This means that, once again, the editor is working with a common translator - using editor command syntax tables instead of object data syntax tables. Naturally the actions taken as a result of analysing syntactically valid commands for the editor are 'editing semantic action routines' rather than those for, say, some programming language compilation. Once this principle of separation is applied to the editor, then it can effectively self-modify itself according to context by, say, detecting that it is being asked to debug a program and insert modified action routines for appropriate commands.



The possibilities of this kind of context sensitivity are, of course, more far reaching than mere editing. The whole question of user adaptability, command language standards, etc rears its head and offers almost boundless opportunity for further research and development using the fundamental tool being developed for this portable system. Any work of this kind is, however, considered to be beyond the project as it is currently defined. There is no intention to progress in this direction without the availability of considerable extra resources in both manpower and equipment.

**Object Data Mapping** This pair of mappings is potentially the most difficult of all - at least for certain kinds of object. It is fairly well known how to carry out syntax analysis and symbol determination for programming, command, editing, etc languages. What is not so well understood is the *unparsing* needed to present human understandable representations of a very wide range of object structures to the user. The tree-like nature of parsing structures and intermediate languages in the compiler system, made it natural to think in tree terms for normal editing. Permanent object storage for objects eventually to be translated by the compilation system is therefore understood to be as a linearised tree embodying the syntactic structure (whatever that may be!) of the object.

One of the advantages of storing as a symbol tree is to avoid unnecessary use of storage space, to avoid unnecessary reparsing (and rereading) of source 'text' and to enable a common storage form to be adopted irrespective of the nature of the object - merely the shape will change. The tree form to be used is therefore a list { of lists \* } of elements. The number of tree levels involved is dependent upon the object, but in practice is unlikely to be more than a dozen or so for most objects being stored.

The need for unparsing means that it should be possible to view one object in more than one way (at once - in different editing windows, say) or, alternatively to move one or more elements of an object into another object with a possibly different structure. Naturally such moves or alternative views should be sensible (for example, converting a loadable program image file into a number of areas in some computer memory - otherwise known as linking). Research into the conversion of a memory binary image into a high-level language source program text has not progressed very far, so such conversions will also be limited to the possible!

**The non-Editing Command** The editor is being designed to incorporate one non-editing command - *run*. This is the most important feature of the whole system, since it enables the editor to run translators (to compile parts/all of programs), interpreters, or really any other program at all - either under close control or 'free' as the user may specify. This of course introduces the testing facilities which are to be built into the project, in particular debugging. As already mentioned briefly, this is only another kind of editing of a dynamically changing object. The editing commands map very naturally onto all debugging commands and the use of multiple windows enables code and data to be 'watched' as execution takes place at whatever 'tree-level' desired. The detailed specification of the necessary debugging actions is not expected to raise any prohibitive difficulties.

### Interpreting

While Rcode was designed to enable an interpreter to 'run' a program which had been translated thus far, it was realised that interpretation of Rcode would be non-trivial. As the design of Peano progressed and the intention to remove all redundant runtime code became uppermost in the concerns about machine code generation, it was realised that interpretation of program elements (e.g. manifest expressions) at compile time was essential for the entire system. In addition to this intention becoming firmer, the addition of the portability criteria had inevitably resulted in more and more code being added. While it was hoped that the machine code generator produced would be considerably better than those seen for Modula-2, Pascal, etc on the VAX, it was seen that the ultimate requirement was to interpret as much at translation time as could be done statically.

A review of all this suggested that not only should it be possible to evaluate manifest expressions at translation time, but also it should be possible to carry out static assignments (initialisation code!) to runtime variables.

In order to carry out all this interpretation, it must be possible for the interpreter to run appropriate library routines already existing in object code form, as well as other modules still in Rcode form. Such an interpreter for Rcode was designed and a considerable amount of code written before the question of size reared its ugly head. This interpreter - or at least nearly all of it which would be needed for translation time evaluation only - was going to be very large indeed - for what it did. Additionally it would have to be co-resident with the remainder of the compiler code during machine code generation.

The question of size suggested that a smaller '*interpreting machine*' would be desirable. Consideration had already been given in looking at the problems of portable machine code generation to a second intermediate language which contained information about the real operand facilities offered by the target machine class of architecture. It was soon realised that knowledge about these operand features was already being considered for

18

the interpreter design, to hold information about the runtime environment. The decision was therefore made to introduce a second intermediate language based around the architecture of a class of real machines and a small instruction set based around the semantics of Rcode. The compiler would need to convert all Rcode to this form after completing high-level optimisations, so that it could readily be interpreted in a far smaller program.

Since almost a complete interpreter is needed for translation time, it is a very small step in design and coding effort to produce a stand-alone interpreter of this second intermediate code.

### Generic Action Sets

The recent interest in minimal (RISC) instruction set computers led to the realisation that, with very few exceptions, the most effective code would be something of this nature - provided that the structure of the original Rcode tree was retained to assist in optimisation at this level. It was therefore decided that if the structure of the tree was retained a very simple linear code based around the notions of a Generic Action Set would suffice.

A generic action is one which is independent of the kind of object on which it is operating - such as *ADD*, *MOVE*, etc. In addition to this feature, a generic action does not depend upon where the operand(s) may be inside a machine. The code developed has come to be called a Generic Action Set code (GAS-code for short) since each instruction represents a set of machine code instructions. The size of such a set varies with individual machines, the GAS-code does not vary in this way. It will vary with different machine architectures to some extent (the register-less machine, for example, will have different operand structures).

The set of codes in GAS-code has turned out for the machine architectures being considered initially to have 32 instructions. A careful survey of the instruction sets of both of the initial target machines shows that, for the VAX all but 9 out of some 253 instructions may be produced by the code generator (these are all kernel special), while for the Zilog machine only 7 instructions (relating to kernel multi-processors) may not be generated. This is considered to be satisfactory for a first production implementation, particularly since the interpreter will be quite small.

The original design of Rcode included a facility for extending the code to have other operations - as two-byte codes. One of the extension bytes is *Reserved*. This is used as the first byte of GAS-code, which can therefore appear on the Rcode tree, along with area declarations, literal values, etc. There is therefore no additional overhead in compiler code to manipulate a GAS-fitted tree!

Interpretation of a GAS-fitted tree within the compiler code generation mechanism is merely, therefore, one of interpreting some sub-tree and replacing it by a literal result or an initial variable value. Once this has been done then major redundant code elimination and tree swinging optimisation can be done while carrying out preliminary resource (register) allocation and setting up jump tables, etc before final code generation.

### Compiler Output Code

In addition to binary code, constant and variable data areas, the compiler machine code generator must produce directives for a linker such as external module object references, system object references, relocation information, debugger information, etc.

**Portable Linker** While it would seem sensible to have only one linking process on any one computer, it has been found necessary to introduce a second - portable - linking program. This is to be used before the native system linking is carried out. The main reasons for doing this is to

- a. Minimise the number of objects included from imported modules. Many linkers for well-known machines include entire modules if any object is referenced, rather than merely those objects needed - whether routine or variable!
- b. Detect circularity and arrange the correct order of initialisation of module code. Considerable problems have been created by both of the Modula-2 implementations seen, in either not detecting circularity at all (and having tremendous sequences of initialisation calls) or not using a *weakest precondition* algorithm.
- c. Provide a linker for many small microcomputers which do not have one!

### Conclusion

The extension of the original rather limited project to satisfy immediate student needs has inevitably led to considerable delay. The benefits of attempting to design truly portable systems are evident. The remaining problems of implementation do not appear to be insurmountable and the design for portability principle should certainly be applied far more widely than is currently used.



While there are inevitably likely to be minor design changes as detailed implementation progresses, no major change is foreseen to the overall project structure described here. The use of Modula-2 as the implementation language has certainly offered ways of implementing certain features without which this project would have inevitably failed or had to await a bootstrap implementation. Even with the existing system, many changes have had to be made to the original compiler in order to correctly generate the portable software.

The idea of the Editing Environment as a kind of user interface is now central to the project, whereas originally only compiler portability was being addressed.

The use of compiler translation facilities - and 'output' activities for whatever may be the kind of object being edited is the ideal end result. For the foreseeable short term future, however, it may not be possible to control all translations/outputs through the command environment. It is likely that too much code would need alteration to be practical in the beginning. If this portable environment concept proves attractive to users, however, it is hoped that more translator tables and semantic actions for output will be added to the initial set.

#### Acknowledgements

The project team has varied in size and effort over the years - and will most likely continue so to do. Thanks at the time of writing are due (in alphabetical order) to RM Archer, M Byrne, CJ Francis, A Greer, B Hoult, LI d'Oliveiro, SH Seo and A Sleeman. To them and many other friends and colleagues who have encouraged us when needed - our heartfelt appreciation.

17 Oct 1985

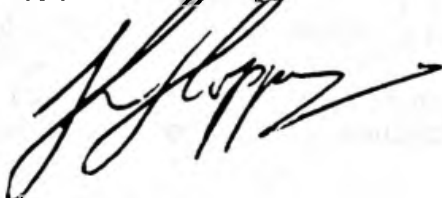
Dear Sir,

Quite by chance recently I was made aware of the existence of your association. Because of our relatively bad introduction to the Modula-2 language - bad and inconsistent implementations that is - we have spent nearly two years in designing and developing a completely portable version - independent of machine or operating system. Our current status is that the entire portable run-time library has been subjected to intensive testing for a year and by February 1986 Version 1.2 - with all known bugs fixed will be available for VAX/VMS. Our UNIX version is now being tested, while preliminary work on an MS/DOS version is underway.

We have produced a bootstrap compiler on the VAX to enable us to produce the portable system. Design has just been completed of the final compiler details and two-thirds has been implemented. As you will appreciate, this is nothing like any existing compiler - which are all severely bug-ridden so far as we can tell (or rather our students keep bringing up new compiler bugs almost every week!).

I enclose a copy of our design rationale which you may wish to disseminate. Full Reference Manuals and User Guides are available for the Portable Library, Rcode and GAS-code (several hundred pages of A5!) should anyone be interested (we have to make a small copy and postage charge).

Yours faithfully



K. HOPPER  
Senior Lecturer

## The ETH-Zürich Modula-2 compiler for the Macintosh by Chris Jewell

Copyright © 1986 by Christopher T. Jewell.

The author grants permission to reproduce this work, provided that this notice is included in each copy, and no consideration is charged for the copies.

Peter Fink and Franz Krönenseder of the Institut für Informatik at ETH Zürich adapted a 68000 Modula-2 compiler to the Macintosh. They also made a prerelease version of the compiler and libraries available on a noncommercial, no support, no guarantee basis. Matthias Aebi put the software onto USENET, the Unix System users' network. The following notes describe one user's experiences with, and observations about, the compiler and libraries.

### Thanks

First, a "thank you" is due to Messrs Fink and Krönenseder for building this software, and making it available. Further, anyone who is going to use the compiler, linker and library should read section 4.1.8C, entitled "History (Acknowledgements)", of the User Guide to the Modula-2 System (file MODLIB:GUIDE.TEXT) so that he will know whom else to thank.

In my case, thanks are due to your newsletter editor, who downloaded the software from USENET, and made it available to those of us who do not have access to either USENET or typical academic software distribution channels. Thanks, also, to Computer Plus, of Sunnyvale CA, who graciously allowed me to run the benchmark on a Macintosh Plus, as soon as they received one from Apple.

### What it is

The three diskettes which make up the ETHZ Modula-2 implementation for the Mac give one the means to compile, link, and run Modula-2 code which is comparatively portable. Several interesting sample programs are supplied: they probably look very much the same running on a Mac as they would running on various 68000-based systems developed at ETH (including user dialog, variable names, and program comments in German). The documentation with the copies circulating in the US is in English. Library modules give access to the parts of the Macintosh toolbox which the implementors needed in order to get their software working (notably QuickDraw, EventManager, and the standard file dialog package).

### What it is not

This is not a commercial system for developing stand-alone applications for the Macintosh. Most of the Macintosh Toolbox routines are not (yet) supported. A ready-to-run object program produced by the linker is not a Macintosh "application", but rather a "document" whose associated



"application" is the runtime support package (probably named EXEC512K on your disk). When you double-click on an M2 program, the Finder will launch the runtime support, which will in turn run the program.

"Glue" for the Macintosh ROM routines (This section assumes familiarity with Apple's *Inside Macintosh*, and the Macintosh Development System, or MDS.)

If you want to use Macintosh OS or Toolbox routines which are not already supported by the supplied library modules, you will have to write your own glue procedures, using the CODE feature of the compiler, to get to the A-line traps. Even the stack-oriented Toolbox traps will require non-trivial CODE procedures: you cannot just use an A-line trap word with the auto-pop bit turned on. When writing your CODE procedures, you will need to take account of the following facts.

1. The compiler generates code which makes it the responsibility of the calling routine to drop the parameters from the stack, while the Mac ROM routines consider it the responsibility of the called routine. Therefore, an interface to a ROM (or other Mac-type) routine must duplicate the parameters supplied by the caller (including space for the result if you are calling a function), before issuing the \$Axxx trap, then copy the function result to where the caller expects it after the trap and before returning to the caller.

2. The M2 system uses register A5 to reference the current M2 PROCESS. This is incompatible with the use of A5 expected by some of the ROM routines, which expect to find a pointer to QuickDraw global variable thePort at the location referenced by A5. An interface routine must save A5, and load it from system global variable currentA5, which is at location 904H, before the trap, and restore M2's value afterwards.

3. The QuickDraw definition module contains an incorrect declaration for TYPE Pattern. The Lisa Pascal declaration is

TYPE Pattern = PACKED ARRAY [0..7] OF 0..255;

while the supplied Modula-2 declaration is

TYPE Pattern = ARRAY [0..7] OF [0..255];

Since the compiler doesn't pack subrange types, each element of the array gets 2 bytes, and the whole array gets 16 bytes, rather than 8, as it should. Try

→ TYPE Pattern = ARRAY [0..7] OF SET OF [0..7];

4. The definition module for QuickDraw has the global variables commented out, with the comment that those variables are available in the surrounding environment, but the comment doesn't tell you how to get at them. You need to declare a pointer to a record containing, as fields, all the QuickDraw global variables, in reverse order (because variables are allocated from the highest address down, while fields in a record are allocated from the bottom up). Then set the pointer to the contents of the address to which currentA5 is pointing, less 7EH. For example:

```

VAR currentA5 [904H] : POINTER TO ADDRESS;
    grafGlobals      : POINTER TO RECORD
        randSeed : ADDRESS; (* use as LONGINT *)
        screenBits : QuickDraw.BitMap;
        arrow      : QuickDraw.Cursor;
        dkGray     : Pattern;
        ltGray     : Pattern;
        gray       : Pattern;
        black      : Pattern;
        white      : Pattern;
        thePort    : QuickDraw.GrafPtr;
    END (* grafGlobals^ *);
...
grafGlobals := currentA5^ - 7EH;

```

5. Contrary to the implications of the *User's Guide*, the compiler option \$P- seems to work only for the very next procedure encountered. If you want to have a whole bunch of code procedures, for example as interfaces to Macintosh ROM routines, you must repeat (*\$P-*) before each procedure.

6. As far as I can figure out, the compiler seems to generate code to check for stack overflow at the start of some procedures, but not others, independent of the (*\$S-*) option. Perhaps stack checking occurs whenever there are value parameters of structured types, which must be copied by the procedure entry code. (?)

An example of the required "glue" is included at the end of this article (if the editor had room for it.)

#### Language supported, features, and implementation restrictions

The compiler is a descendant of the original five-pass compiler for the PDP-11. The output of the compiler must be processed by the supplied static linker, a very slow program.

1. CARDINAL, INTEGER, BITSET, and WORD are two bytes.
2. REAL is 4 bytes, and does not use the Macintosh SANE routines.
3. Base types of sets are limited to at most 16 elements. (No SET OF CHAR).
4. LONGxxxx types are not supported. However, all operators which may be applied to two CARDINALs (including <, <=, >, and >=) may also be applied to two ADDRESSES, so that ADDRESS may be used as a substitute for LONGCARD. The supplied library modules support the ability to do formatted reading and writing of ADDRESSES, just as for CARDINALs.
5. Open array parameters are limited to VAR parameters only. However, the compiler permits you to pass a string literal actual parameter where the corresponding formal parameter is declared as VAR ARRAY OF CHAR. You may pass anything to a VAR ARRAY OF WORD formal parameter.

6. The volume which contains the compiler overlays and the module library must be named MODLIB. I'm just glad that I don't have any other software that is sensitive to volume names, or I'd have to spend a lot of time renaming my hard disk.

7. Aside from the usual stuff, the pseudo-module SYSTEM supports direct access to 68000 registers, as well as an arithmetic shift operation on word-sized types and addresses.

### Error Detection

The code generated by this compiler is deficient in run time error detection, by the standards which Professor Wirth espouses in reference [1]. In particular, the compiled code failed to detect any the following programming errors:

1. Assignment of an out-of-range value to a variable of a subrange type, even for the easy case which is supported directly by the CHK instruction of the 68000, that is:

TYPE SubType = [0..n]; (\* 0 < n < 32768 \*)

2. Assignment of a negative INTEGER value to a CARDINAL variable, or a CARDINAL value greater than CARDINAL(MAX(INTEGER)) to an INTEGER variable. Assuming the value to be first computed in a data register Dn, this test could be performed very efficiently by generating the instruction  
CHK Dn,Dn

3. Overflow on INTEGER or CARDINAL arithmetic. (Note that for INTEGER arithmetic, the very inexpensive TRAPV instruction would do the job.)

4. CHR or VAL, where the cardinal argument is greater than ORD(MAX(t)), where t is the destination type.

5. The value used for NIL is 0FFFFFFFH. Therefore, the 68000 will detect a NIL pointer reference if the pointer is to a word-aligned type. However, a NIL value in a pointer to a CHAR, BOOLEAN, or small enumeration type, or a reference to a field of one of those types in a record, will not cause a run-time error.

On the other hand, the compiled code will detect both indexing outside an array, and stack overflow at procedure entry time. The latter is much better than the usual Macintosh scheme of checking on each vertical retrace interrupt to see whether or not the stack and heap are now overlapped. What can the Apple engineers have been thinking of when they settled on such a probabilistic scheme for detecting stack overflow?

In short, while the compiler produces a correct object program from a correct source program, don't expect too much help in finding your mistakes. Errors which are detected usually throw you into Macsbug (if you have it installed), or



generate a bomb. There does not appear to be any provision for connecting the error traps to a coroutine.

### Performance

The benchmark in Appendix 2 of reference [1] was run on a Lisa 2+5 with MacWorks, with the results shown below. Lilith performance numbers from reference [1] are shown for comparison. In each case, the number shown is the number of iterations of a loop performed in one minute, so bigger numbers are better.

#### Note:

1. The effective clock rate of the 68000 MPU in the Lisa is about 5 MHz (slightly less due to occasional wait states to keep the MPU in step with the refresh circuitry). If a similar compiler were tested on another, faster 68000 system, such as a 10 or 12 MHz Stride, the performance to be expected can be estimated as the ratio of the clock rates (that is, the effective rates, allowing for wait states, refresh time, etc). It appears that a 10 MHz 68000 system should match or beat a Lilith for speed in most tests, and a 12 MHz 68000 should beat a Lilith hands down for everything except REALs (tests e and f).

2. The times for a Mac 512K are probably exactly the same as those for a Mac Plus. The performance enhancements of the Mac Plus over the earlier Mac are all in the area of either I/O or the ROM routines, neither of which are reflected in the Lilith benchmark used here.

3. The ratio between the Lisa and Mac+ numbers is not the same for all tests, which may indicate observation errors by the author. Caveat lector.

Now let's benchmark a 68020, an NCR ALP, an Inmos Transputer, and an Acorn RISC Machine! Seriously, of course, speed must be weighed against such other considerations as the superior error detection of the Lilith.

facility	Lisa	Mac+	Lilith	Lilith/Lisa ratio
a empty REPEAT loop	245	269	321	1.31
b empty WHILE loop	246	269	334	1.36
c empty FOR loop	283	313	422	1.49
d CARDINAL arithmetic	81	104	187	2.31
e REAL arithmetic	23	27	130	5.65
f sin, exp, ln, sqrt	12	14	87	7.25
g array access	62	68	109	1.76
h same with bounds test	49	55	89	1.82
i matrix access	101	120	197	1.95
j same with bounds test	83	98	164	1.98
k call of empty procedure	71	74	144	2.03
l with 4 parameters	52	55	94	1.81
m copying arrays	49	53	63	1.29
n access via pointer	62	66	125	2.02

## Glue Samples

The following fragments show typical CODE constructs for accessing the Macintosh OS and Toolbox.

```
IMPLEMENTATION MODULE MemoryMgr; (* 20 Aug 1985 -- C.Jewell *)
```

```
FROM SYSTEM IMPORT ADDRESS, CODE, REGISTER;
FROM SysTypes IMPORT LONGINT, Handle;
```

```
(* The following constants are for use with the CODE
 * construct. They embody the instructions which will
be * needed to provide an interface to the ROM routines.
 *)
```

```
CONST
  Mac      = 0A000H; (* Macintosh OS or Toolbox trap*)
  Mac1     = 0A100H; (* OS trap with "pass A0" set *)
  DupWParm = 03F2FH; (* MOVE.W d(SP),-(SP) *)
  DupLParm = 02F2FH; (* MOVE.L d(SP),-(SP) *)
  GetAParm = 0206FH; (* MOVEA.L d(SP),A0 *)
  GetDParm = 0202FH; (* MOVE.L d(SP),D0 *)
  StoreA0  = 02F48H; (* MOVE.L A0,d(SP) *)
  StoreD0  = 02F40H; (* MOVE.L D0,d(SP) *)
  LEAParm  = 041EFH; (* LEA d(SP),A0 *)
  SaveA5   = 02F0DH; (* MOVE.L A5,-(SP) *)
  SetA5    = 02A78H; (* MOVE.L short.abs,A5 *)
  currentA5 = 00904H; (* an anchored variable *)
  RestoreA5 = 02A5FH; (* MOVE.L (SP)+,A5 *)
  CopyResultW = 03F5FH; (* MOVE.W (SP)+,d(SP) *)
  CopyResultL = 02F5FH; (* MOVE.L (SP)+,d(SP) *)
  MakeRoomW  = 0558FH; (* SUBQ.L #2,SP *)
  MakeRoomL  = 0598FH; (* SUBQ.L #4,SP *)
  RTS        = 04E75H; (* return from subroutine *)
```

```
(*SP-*)
(* Here is a register-based routine. The trap takes its *)
(* parameter in D0, and returns its function result in A0 *)
PROCEDURE NewHandle (logicalSize : Size) : Handle;
```

```
  BEGIN
```

```
    CODE (SaveA5);
    CODE (SetA5); CODE (currentA5);
    CODE (GetDParm); CODE (8);
    CODE (Mac+34);
    CODE (StoreA0); CODE (12);
    memError := INTEGER (REGISTER (0));
    (* Note that memErr is an integer variable, *)
    (* exported from our definition module *)
    CODE (RestoreA5);
    CODE (RTS);
```

```
  END NewHandle;
```

```

(*$P-*)
(* Here's another register-based trap... It wants the handle
in *)
(* A0, and returns in D0 either a positive size, or a
negative *)
(* error code. Since the compiler doesn't have a signed 32-
bit *)
(* type, we will have to compare for >= the least ADDRESS
value *)
(* which has its sign bit turned on. *)
PROCEDURE GetHandleSize (h: Handle) : Size;

```

```

    BEGIN
        CODE (SaveA5);
        CODE (SetA5); CODE (currentA5);
        CODE (GetAParm); CODE (8);
        CODE (Mac+37);
        CODE (StoreD0); CODE (12);
        IF REGISTER (0) >= 80000000H THEN
            memError := INTEGER (REGISTER (0));
        ELSE
            memError := 0;
        END;
        CODE (RestoreA5);
        CODE (RTS);
    END GetHandleSize;

```

```

END MemoryMgr.

```

```

IMPLEMENTATION MODULE ResourceMgr;

```

```

FROM SysTypes IMPORT Handle, OSStr255;
FROM SYSTEM IMPORT CODE;

```

```

    CONST
        MOVEQ = 07000H; (* MOVEQ #0,D0 (just add data) *)
        DBRA = 051C8H; (* DBF D0,disp *)
    (* Many of the constants shown in MemoryMgr are also needed
    *)

```

```

(*$P-*)
(* Here is an example of a stack-oriented function trap. *)
PROCEDURE OpenResFile (VAR fileName : OSStr255) : INTEGER;
    BEGIN

```

```

        CODE (SaveA5);
        CODE (SetA5); CODE (currentA5);
        CODE (MakeRoomW);
        CODE (DupLParm); CODE (10);
        CODE (0A997H);
        CODE (CopyResultW); CODE (12);
        CODE (RTS);
    END OpenResFile;

```



(\*\$P-\*)  
 (\* And here's another, with more parameters, \*)  
 (\* but no function value to return \*)  
 PROCEDURE GetResInfo (theResource : Handle;  
                       VAR theID : INTEGER;  
                       VAR theType : ResType;  
                       VAR name : OSStr255);  
 BEGIN  
     CODE (SaveA5);  
     CODE (SetA5); CODE (currentA5);  
     CODE (MOVEQ+3);  
     CODE (DupLParm); CODE (20); (\* done 4 times \*)  
     CODE (DBRA); CODE (OFFFAH); (\* to the DupLParm \*)  
     CODE (0A9A8H);  
     CODE (RTS);  
 END GetResInfo;  
  
 ...  
 END ResourceMgr.

### References

1. N. Wirth, The Personal Computer Lilith, Institut für Informatik, ETH Zürich, April 1981. Reprinted October 1983 by Modula Research Institute, Provo, Utah.

## **NewStudio™: Engineering a Modula-2 Application for Macintosh™**

Andrew H. Davidson, H. Bruce Herrmann, Erin Rae Hoffer

*NewLine 7*  
P.O. Box 1211  
Culver City, CA 90232  
213/277-7217

*Abstract* NewStudio™ is a 3-D graphics application for visual designers of all kinds. The initial implementation, in Modula-2 for the Apple Macintosh™, is tailored for use by architects. NewStudio™ allows a designer to rapidly explore alternative designs in the early stages of a project. It accomplishes this by generating 3-D views of the design, in a flexible manner and with a variety of rendering methods, as it is being created.

This paper describes the software engineering aspects of the design and implementation of NewStudio™, as they are related to Modula-2 and the Macintosh™. We discuss the implementation language and compiler selection decisions, the development environment, and the software architecture of the application. Finally, an analysis of the project is made, considering the facilities and problems encountered during development.

### **Introduction**

In a typical architectural project, the building design phase is limited to approximately 20% of the total time allocated to the project, due to budgetary and deadline constraints. The remaining 80% of the time is used for the creation of "working" (construction) drawings. This requires the architect to conceive and develop the overall plan of the project in a fairly short period of time. While computer aided design systems can help speed up the production and modification of construction drawings once the design is complete, they are not normally an effective tool in the design process itself. They impose a rigid and inflexible methodology upon the designer, which is a reasonable price to pay for sets of accurate detailed production documents but entirely unreasonable for more preliminary design documents.

Architectural designs are generally conceived and drawn in two views - plan (top) and elevation (side). One of the most difficult tasks for the designer is converting these two 2-D views into a realistic 3-D rendering. However, this is an important part of the process because only in 3-D can the effectiveness and appropriateness of a design be evaluated. But perspective, and even parallel projection renderings, are difficult and time-consuming to execute, and thus slow down the design process, shortening even more an already tight schedule. NewStudio™ is a software tool which allows a designer to rapidly explore alternative designs in the early stages of a project, when he is experimenting with rough shapes and layout constraints. The designer enters objects into the system by specifying two 2-D views of the object which are automatically extruded to 3-D solids. NewStudio™ can calculate and present a 3-D view at any time during this process.

The remaining sections of this paper describe the software engineering aspects of the design and implementation of NewStudio™. We discuss the various hardware and software choices made for implementation, the development environment for the project, and the software architecture of the application. Finally, an analysis of the project is made, considering the facilities and problems encountered during development.

### Choices

Once NewStudio™ was conceived, a number of decisions had to be made before its implementation could be begun. First was the choice of a target computer. We felt, for a number of reasons, that NewStudio™ should run on a microcomputer. The primary user of the product is likely to be an architect in a relatively small firm. The current popularity of microprocessors in professional fields, their low cost, and the wide software base available in a variety of applications all contribute to the micro's appropriateness for this task.

Having targeted NewStudio™ for a micro, two obvious candidates presented themselves: IBM PC and Apple Macintosh. We chose the Macintosh for its well known ease of use, its appeal as a graphic tool, and the standard user interface it provides. Despite the richer development environment available on the PC, and the difficulty of developing software for the Macintosh, marketing considerations dictated the choice.

Once we had settled on the Macintosh, we had to select a development language. We wanted to do all of our development work directly on the Macintosh, rather than using a Lisa cross-development system, because of the cost and the uncertain future of the Lisa. At the time we began work on NewStudio™ (December, 1984) the programming languages available on the Macintosh were 68000 assembly language, Pascal, C, and Module-2. We ruled out assembly language immediately because of the high-level nature of our application, and the inherent difficulties in implementing



and manipulating complex data structures in assembly language. Macintosh Pascal is a true interpreter, and not suitable for use as a commercial development system because of memory limitations, unacceptable run-time execution speed, and incomplete access to operating system functions. Although there were a number of mature C compilers on the market, we preferred a language with strong type-checking and had personal biases against the style of programming into which one tends to be led with C. Module-2, being strongly typed, providing a rich set of data structuring and control facilities, allowing efficient and convenient access to system level features, and designed for large system development efforts, was an alternative that met our requirements. None of us had used the language before, but strong Pascal backgrounds and an interest in working in Module-2 led us to feel confident about our choice of Module-2.

We beta-tested two different implementations of Module-2 for the Macintosh. The first was a UCSD p-System product from Volition Systems. The other, MacModule-2, is derived from the original ETH M-code compiler and is marketed by Modula Corporation. We decided to use MacModule-2, partly because it was closer to being a commercially supported product at the time, and partly due to our inexperience with the p-System environment. We had some concern about compiler and run-time performance of the MacModule-2 system, due to the fact that it is an interpreted system, but preliminary benchmarks indicated run-time performance to be adequate for graphics interactions. As it turned out, the Volition Systems product was never commercially released.

### Implementation

The implementation team for NewStudio™ consisted of three part-time software engineers using independent 512K Macintosh systems for all development and testing. One of us concentrated on the interactive graphics portion of the software - the Macintosh Toolbox interface (drawing routines, mouse input, menu and window control, etc). A second developer did all of the 3-D graphics software - clipping, perspective, and rendering algorithms. The third member of the team implemented a data base management system (DBMS) and various utilities, handled the operating system interfaces, and served as the system coordinator and integrator.

There were a number of learning curves which had to be overcome during the project, and all of these figured heavily in the character of our development efforts. Module-2, although familiar to us in theory, was a new language for all of us to be writing in, and there is no substitute for experience in mastering the details and idiosyncracies of any programming language. Another area which was new to all of us was the Macintosh Toolbox. It is a software package which allows a Macintosh programmer to create very sophisticated applications, but with that power comes complexity. It takes a long time to absorb all of the capabilities available in the Toolbox, and it does not lend itself easily to stepwise refinement - you need to understand most of it before you can

use any of it, and this takes many re-readings of the Toolbox manual (Inside Macintosh). Another learning curve was the MacModule-2 system itself. It provides a large set of library modules, many of which we needed to use, including complete interfaces to the Toolbox routines. And finally, despite having been specified in fair detail beforehand, the design of NewStudio™ itself was changing as we worked on it. Our ideas about user interfaces, necessary features, internal data structures, and implementation techniques were constantly evolving during the development cycle. All of these things conspired to complicate our task.

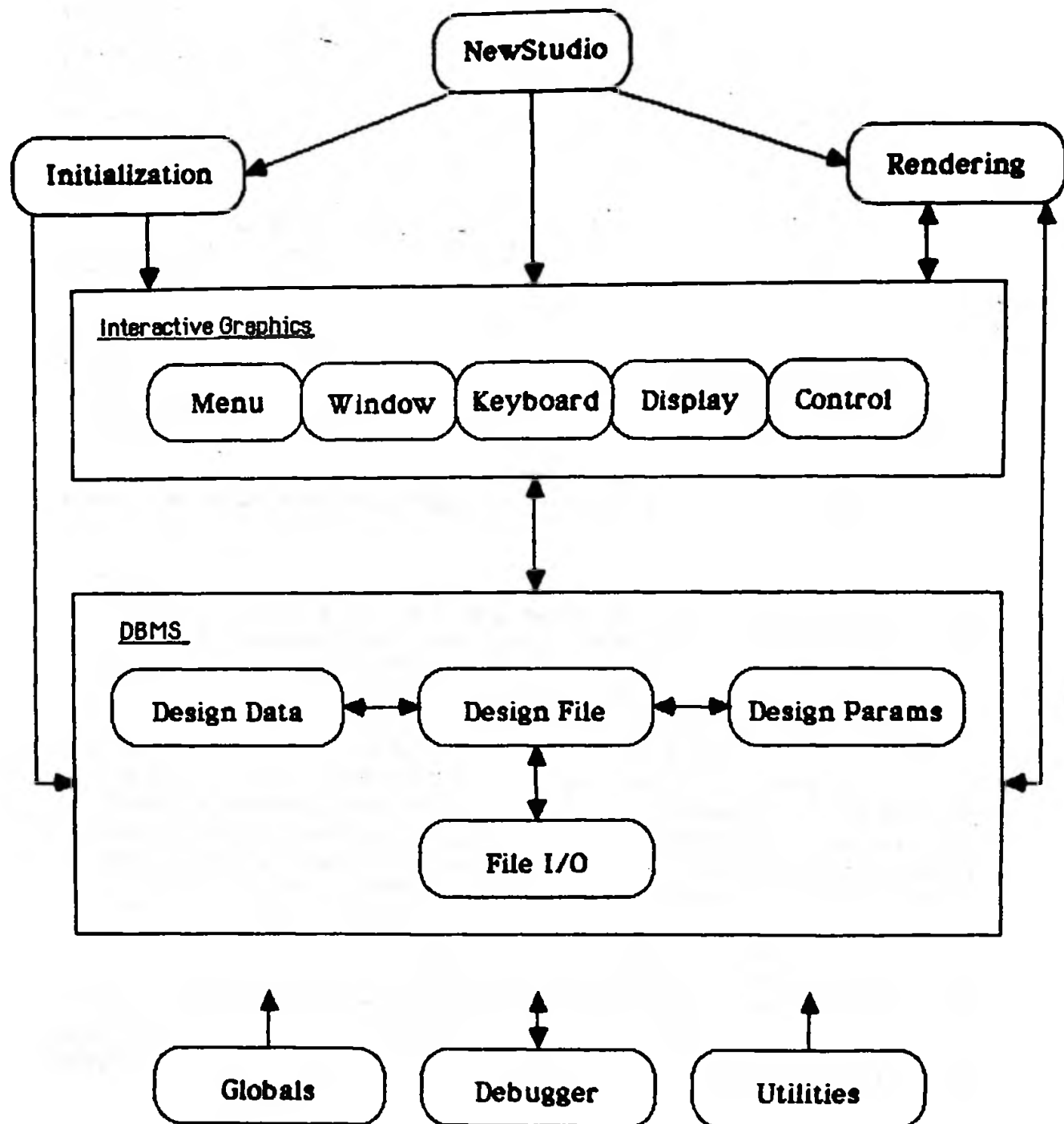
NewStudio™ is divided into 15 Module-2 modules. Figure 1 illustrates this module decomposition. The directed arrows loosely describe the import and export relationships between modules. Module **Globals** contains system-wide constant and type declarations and is imported by all other modules in the system. **Utilities** is a collection of application-independent routines, primarily string manipulation functions (conversion between binary and text format). **NewStudio** is the main program and contains the software which processes all events (mouse, menu, and window activities) dispatching control to all other functions. **Initialization** performs all necessary startup operations for the application and synchronizes the initialization of all other modules. **Rendering** handles all 3-D calculations. The **Menu**, **Window**, **Keyboard**, **Display**, and **Control** modules are very tightly coupled and perform all the user-interface and drawing functions. The **DesignData**, **DesignParams**, and **DesignFile** modules implement interfaces to the different parts of the NewStudio™ data base. Module **FileI/O** concentrates the access to the Macintosh file system in one module.

Because the MacModule-2 system does not have a symbolic debugging capability, we implemented debugging features of our own (in module **Debugger**). This debugger provides the ability to set breakpoints at any place in the system, and to display text messages under program control from these breakpoints. The debugger allows the messages to be directed to a normal text window on the Macintosh screen, a dialog box, a disk file, printer, or serial port (connected to another Macintosh, for example). Whether or not breakpoint messages are issued at all may be controlled dynamically at run time through a special menu item installed when the application enables the debugger. Each breakpoint message is tagged with both the module and routine in which it originates, and this information is automatically inserted into the message when it is displayed. A set of modules which filters messages to be displayed may also be dynamically specified in order to receive debugging messages from specific modules in the system. Figure 2 illustrates the debugger control dialog box, and Figure 3 is a sample breakpoint message.

### **Evaluation**

An evaluation of the software engineering of NewStudio™ with respect to Module-2 must address two areas - the Module-2 language itself, and the MacModule-2 implementation of the language.

## NewStudio™ Module Decomposition





## NewStudio™ Debugger Control

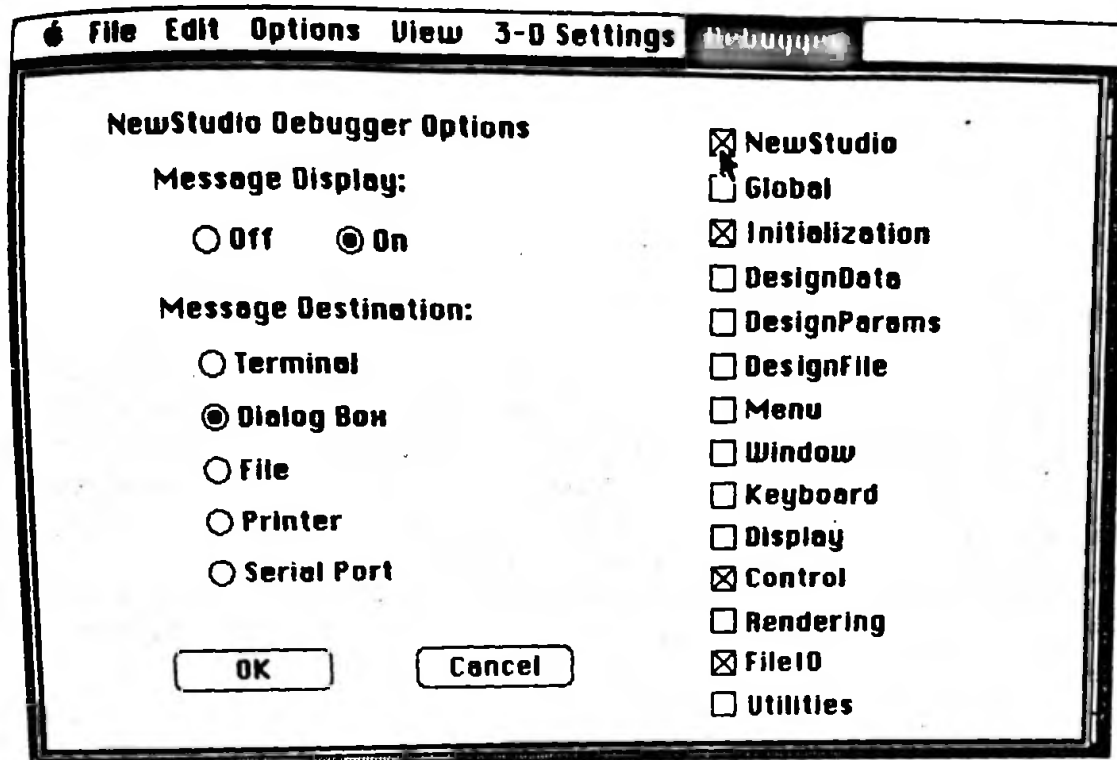


Figure 2

## NewStudio™ Breakpoint Message

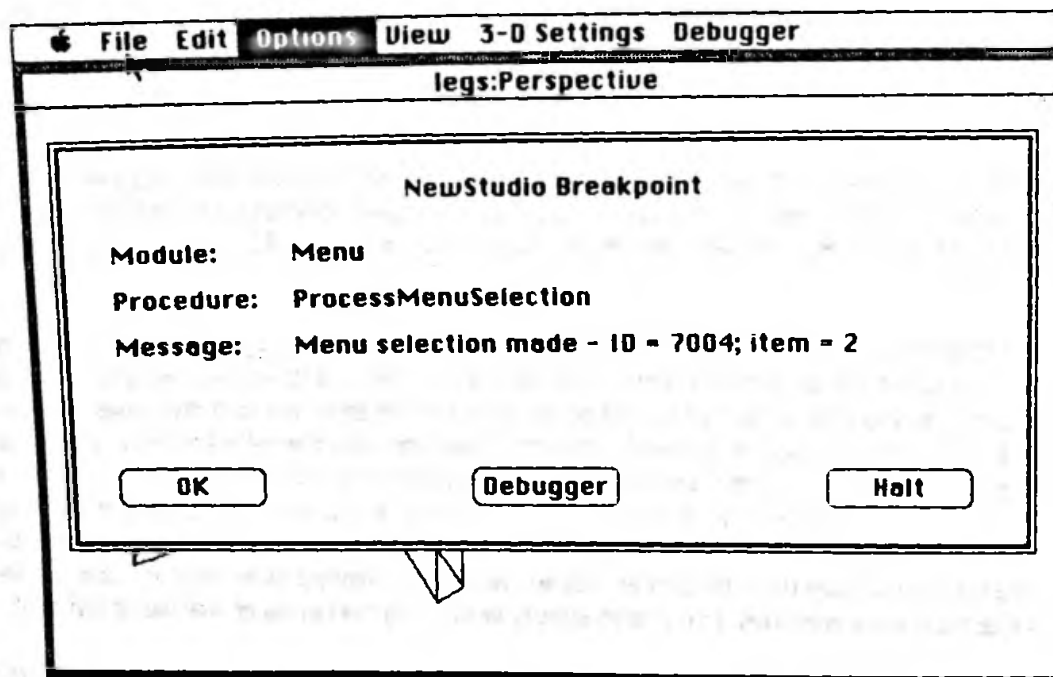


Figure 3

The chart in Figure 4 summarizes the issues.

For the most part, we found that the design and features of Modula-2 worked to our advantage. A strongly typed language enforces a discipline on programming which fundamentally changes the software debugging process: by moving much of that debugging effort from run-time to compile-time. Because of the difficulty in debugging interactive, timing-dependent applications, this is a very useful feature. The rich data-structuring and control mechanisms in the language make it very easy to produce readable, maintainable software. Modula-2 provides clear, flexible, and controlled access to low-level system features on the host computer. In building large, "real-world", applications this is almost always a necessity, and Modula-2 does well in that regard.

Perhaps the most important feature of Modula-2 to us as a development team was the combination of Modula-2's separate compilation facility and the ability to divorce module interfaces from their implementations. Due to the part-time nature of the project, it was necessary to do much of our work at separate locations and times. A typical cycle would be for the development team to spend a weekend together designing and specifying modules and interfaces between them, followed by a week of separate implementation efforts. The following weekend would begin by integrating our modules and then go back into the interface/implementation cycle. The facility provided by Modula-2 for compiling DEFINITION modules to specify interfaces separately from their IMPLEMENTATIONS was invaluable in our "distributed" programming environment.

There were a few things about the Modula-2 language which did hinder, rather than aid, our efforts. The language does not provide for automatic type transfers and conversions in arithmetic expressions. This often forced what would ordinarily be fairly simple mathematical expressions to become long and cumbersome. The underlying formula became obfuscated by multiple layers of CARDINAL to INTEGER, INTEGER to CARDINAL, REAL to INTEGER, INTEGER to REAL, etc. conversions.

The fact that input/output functions are not defined in the language is both a blessing and a curse. It is good in that a system designer using Modula-2 is not limited to the I/O features provided in the language, but is free to take advantage of all the capabilities of the host computer via library modules. However, the lack of a simple intrinsic function like Pascal's `writeln`, allowing the printing of an arbitrary list of mixed data types, is a handicap to efficient debugging efforts. Most Modula-2 implementations provide WriteCard, WriteInt, WriteReal, WriteStr, and WriteLn, and it is true that the function of Pascal's `writeln` can be achieved with these substitutes. Having to import these routines from the proper library and simply having to write more code allows for more typos and more compiles. This makes simply examining the values of variables difficult.

We are well aware of the reasons that the above-mentioned features are not part of the Modula-2 language. We only want to point out that their lack has been a disadvantage to us in our development

## NewStudio™ Software Engineering Evaluation

### Pros

+

### Modula-2

type checking  
code readability  
low-level system access  
separation of DEFINITION and  
IMPLEMENTATION

### Cons

-

type transfers cumbersome in  
arithmetic expressions  
no simple intrinsics for debugging  
(e.g. Pascal WRITELN)

### MacModula-2

simple, efficient Toolbox access  
good run-time performance  
excellent technical support

compilation speed  
M-code interpreter makes using  
native debuggers difficult

Figure 4



environment.

The MacModule-2 implementation provides simple, efficient access to the Macintosh Toolbox. The run-time system, despite being an M-code interpreter, provides acceptable performance for most functions. We also received excellent technical support from the Modula Corporation. One problem with the M-code interpreter, however, is that it prevents using the native Macintosh debuggers since they operate on the interpreter rather than the actual Modula-2 application code. But the single biggest obstacle to development progress we encountered in using MacModule-2 was the slow compilation speed - approximately 80 lines per minute. This could have been improved somewhat by using a hard disk instead of floppies for source code and library modules, but the real limiting factor in compilation is computation, not I/O. With an average implementation module size of 800 lines, the turnaround time is excessive for large system development efforts. The reason for the slow speed of the compiler appears to be the interpreter overhead, since the compiler itself is written in Modula-2 and runs under the M-code interpreter. Presumably, a native-code version of the same system would be more satisfactory.

### Summary

To date (September, 1985), we have completed a prototype implementation of NewStudio™. It is composed of 15 modules with 10,000 lines of source code. This effort involved approximately 18 person-months in 6 elapsed months. Our estimates for turning the prototype system into a final product are an additional 5-10,000 lines of code with another 12 person-months of effort. In addition, we will be converting computationally expensive portions of the rendering algorithm to 68000 assembly language. Figures 5 and 6 are sample screens of the NewStudio™ prototype in use.

A set of software tools, in addition to a compiler, is necessary for efficient software development. A run-time debugger should be part of every software development environment. It should be symbolic, so that debugging can take place at the source code level, and it should be interactive, so that the developer can manipulate the execution of the program as he is debugging it. A compiler cross-reference is useful in tracking down logical errors in syntactically valid source code. As an example, consider the case of (erroneously) multiply declared variables at different scope levels. A batch "script" file processor helps automate the compiling and linking of systems composed of many modules. These software tools are all independent of the development language.

One final useful tool, specific to Modula-2, is a module dependency analyzer. Because of the version checking across modules which is done by Modula-2 compilers and linkers, it is often necessary to derive a list of modules (DEFINITION and IMPLEMENTATION) which must be recompiled when another module has been recompiled. These dependencies quickly become very

## NewStudio™ 2-D Views

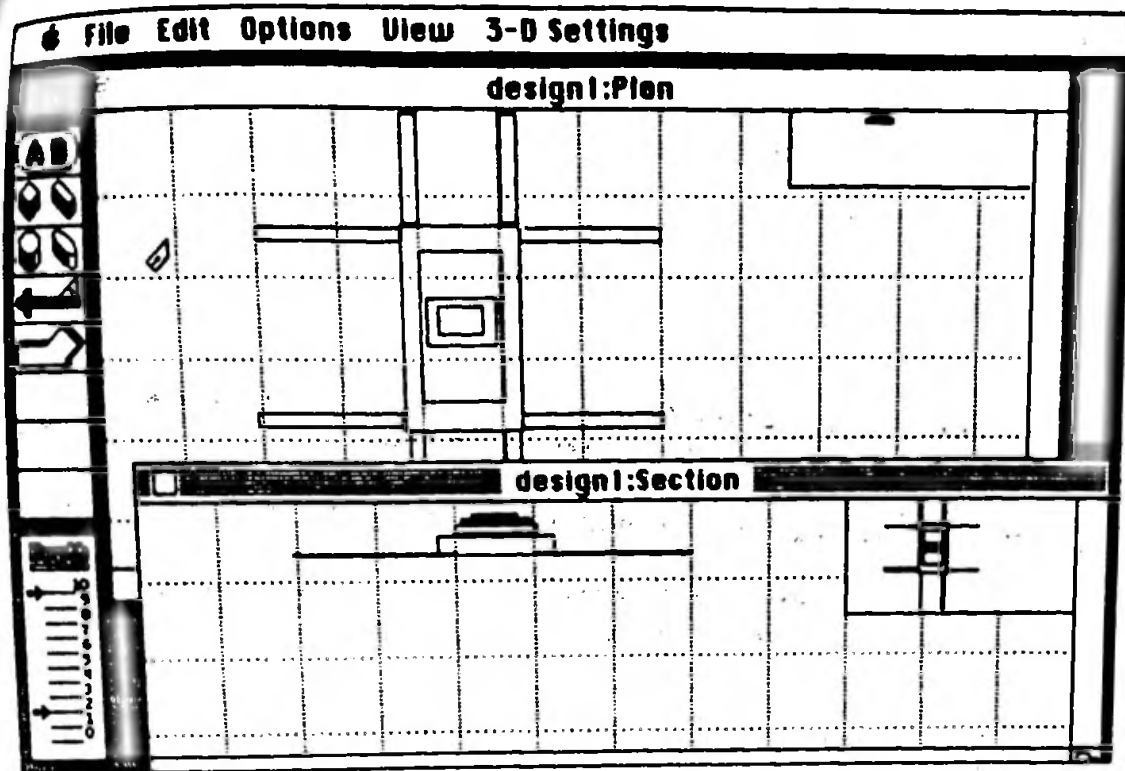


Figure 5

## NewStudio™ 3-D Perspective View

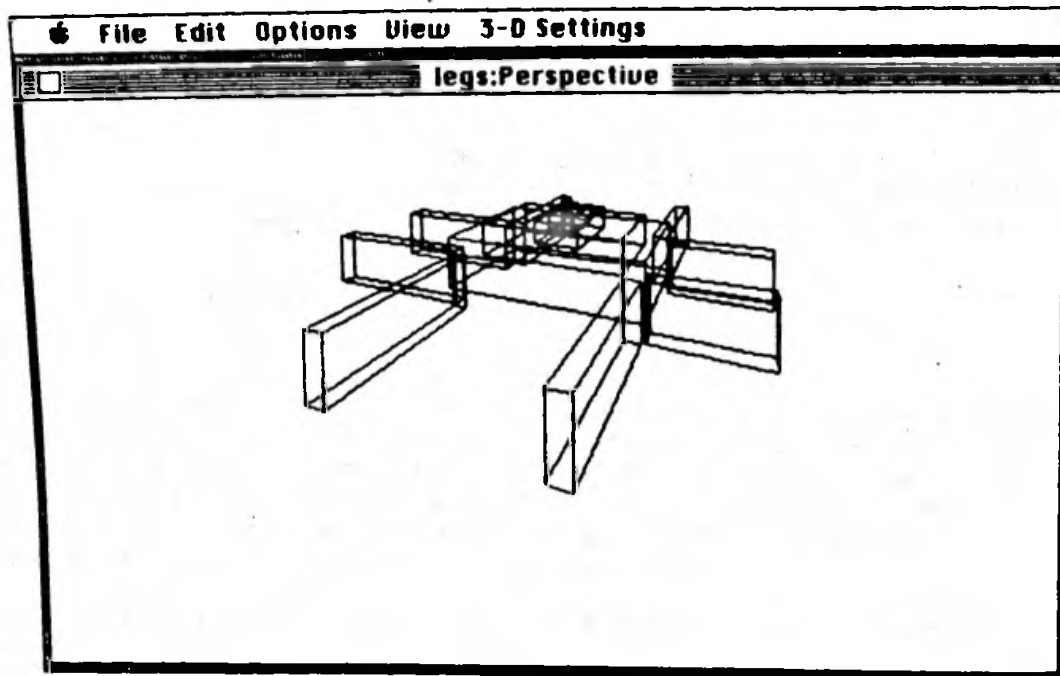


Figure 6

In conclusion, we consider our use of Module-2 as a system implementation language to have been quite successful in the development of NewStudio™.

We wish to thank Donald G. Leeper for his help in unlocking the Toolbox and writing demo programs, and Emily K. Nagle for editing this paper. Mary Ann Morris Ahearn and Stephen M. Skinner were instrumental in the original design of NewStudio™. Finally, for general partnership, continuous support, and NewStudio™ design, we are indebted to Jackson Calvin Green and Emily K. Nagle.

"IBM" is a registered trademark of International Business Machines Corp.

Modula-2 News  
 Editions  
 Spec. of S  
 Modula-2  
 Modus Me  
 Modula-  
 Modu



Modula-2 News # 0 October 1984

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

Modula-2 News # 1 January 1985

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

MODUS Quarterly # 2 April 1985

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

MODUS Quarterly # 3 July 1985

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

MODUS Quarterly # 4 November 1985

MODUS Meeting Report by Bob Peterson  
A Writer's View of Conf, Sam'l Bassett  
Concerns of a Programmer, Dennis Cohen  
Mods to Standard Lib, Nagler & Siegel  
Std Lib and Ext'n to Modula-2, Odersky  
Std Lib for Unix by Morris Djavaheri  
Impl of Std Lib for PC's, Verhulst  
M-2 Compilation and Beyond, Foster  
Modula-2 Processes, Roger Henry

MODUS Quarterly # 5 February 1986

Export Module Identifier, Cornelius  
multi-dimensional open arrays, Wirth  
DIV, MOD, /, and REM, Niklaus Wirth  
Multi-dimensional open arrays, Steiger  
NULL-terminated strings, Poulsen  
ISO Ballot Results re BSI Modula-2  
Draft BSI I/O Library, Eisenbach  
Portable Language Rationale, Hopper +  
ETH-Z Modula-2 for Macintosh, Jewell  
NewStudio: for Macintosh, Davidson +

MODUS Administrators supply single copies at \$5 US or 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 troff, Script and PostScript 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 your submission is permitted. Correspondence not for publication should be PROMINENTLY so marked.

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

## Modula-2 Users' Association MEMBERSHIP APPLICATION

Name : \_\_\_\_\_

Affiliation : \_\_\_\_\_

Address : \_\_\_\_\_

Address : \_\_\_\_\_

State : \_\_\_\_\_ Postal Code : \_\_\_\_\_ Country : \_\_\_\_\_

Phone : (\_\_\_\_) \_\_\_\_ - \_\_\_\_\_ Electronic Addr : \_\_\_\_\_

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

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

Implementation(s) Used : \_\_\_\_\_

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

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

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

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

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

Postfach 289  
CH-8025 Zürich  
Switzerland

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

## Modula-2 Users' Association MEMBERSHIP APPLICATION

Name : \_\_\_\_\_

Affiliation : \_\_\_\_\_

Address : \_\_\_\_\_

Address : \_\_\_\_\_

State : \_\_\_\_\_ Postal Code : \_\_\_\_\_ Country : \_\_\_\_\_

Phone : (\_\_\_\_) \_\_\_\_ - \_\_\_\_\_ Electronic Addr : \_\_\_\_\_

Option : \_\_\_\_ Do NOT print my phone number in any rosters

or : \_\_\_\_ Print ONLY my name and country in any rosters

or : \_\_\_\_ Do NOT release my name on mailing lists

Application as: **New Member** \_\_\_\_ or **Renewal** \_\_\_\_

Implementation(s) Used : \_\_\_\_\_

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

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

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

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

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

Postfach 289  
CH-8025 Zürich  
Switzerland

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



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

MODUS  
Postfach 289  
CH-8025 Zuerich  
Switzerland

Return Postage Guaranteed