The Oakwood Guidelines for Oberon-2 Compiler Developers

Scope

This document is a companion document to the ETH Oberon-2 Report and contains clarifications, extensions, implementation recommendations and a basic library definition.

Purpose

To document the discussions held at the Oakwood Conference in Croydon 1993 and to provide practical guidance for compiler writers. The objective being to have a common approach to Oberon-2 compiler implementations over a wide variety of platforms and to encourage consistency wherever practical.

Authors

See Appendix 2

Revision: 1A First Issue
Edited by:Brian Kirk, Robinson Associates

Dedication

This document is dedicated to the memory of Nick Walsh.

His sound advice and subtle wit combined with

intellectual clarity will be sorely missed by

friends, colleagues and students.

Preface

The Oberon language - together with the Oberon System - was designed and implemented by Prof. Niklaus Wirth and Prof. Jrg Gutknecht at ETH Zrich from 1985 to 1987. Since then it has been used intensively for teaching but also for projects at ETH and elsewhere. After some minor changes - which also led to Oberon-2 - the language finally became stable and mature. Currently, it is available on practically all modern platforms. All these implementations support the same language and even the same interfaces to files, windows and other operating system resources. One could thus speak of a de facto standard.

This was the situation when a group of about 30 compiler developers and vendors met at the Oakwood Hotel in Croydon in June 1993 to agree on a common set of language features and library modules that should be provided by every Oberon-2 system.

This group worked in a very efficient way avoiding bureaucracy and lengthy meetings. Within a few months they produced a document which subsumes the results of the Oakwood meeting and establishes a set of guidelines to compiler developers.

Beside some clarifications to the language report and a modest set of possible extensions, the central part of this document gives hints to compiler developers and defines a basic module library that should come with every Oberon-2 implementation.

I hope that future developers of Oberon-2 systems will stick to these guidelines for the benefit of uniformity and portability. Oberon-2 will only be successful if it does not repeat the mistakes of Modula-2 implementations where a lack of agreement between early compiler developers led to incompatibilities and an all too lengthy standardisation process.

My special thanks go to Brian Kirk and Euan Hill who acted as conveners of the Oakwood meeting and later undertook the difficult task of collecting and assembling all the comments, suggestions and wishes into a consistent and reasonably short document. Thanks also to all individuals listed at the end of this document who contributed in a spirit of cooperation.

Hanspeter Mössenböck

ETH, Zürich

November, 1993

Notes from the Editor

Here is the first issue of the Oakwood Guidelines. It is based on the draft circulated by Email which has been amended based on your feedback and reviewed with Prof. Mössenböck and Josef Templ at ETH. I am aware that possibly all the contributors may possibly be disappointed ! The reason is simple, there have been many ideas put forward and I have tried to include only items which were discussed at the Oakwood meeting or have really strong support. Items in brackets << like this >> highlight topics that require further clarification. Any errors in the draft are likely to be mine.

What next ? My feeling is that the compiler developers (see Appendix B) should try to refine and agree the contents of this document, and not add any more to it, maybe even remove items from it. Above all we should avoid a repeat of the Modula-2 standardisation story as probably nothing useful would be achieved in practice.

Your feedback on the content of the draft and possibility for a further meeting would be most welcome.

Brian Kirk

Robinson Associates
Red Lion House
St Mary’s Street Painswick GLOS
GL6 6QR
Voice (+ 44) (0)452 813 699
Fax (+ 44) (0)452 812 912
e-Mail : robinsons@cix.compulink.co.uk.

CONTENTS

[THIS PAGE INTENTIONALLY LEFT BLANK]

The Oakwood Guidelines for Oberon-2 Compiler Developers

1.0 Introduction

1.1 The Oakwood Guidelines

These guidelines have been produced by a group of Oberon-2 compiler developers, including ETH developers, after a meeting at the Oakwood Hotel in Croydon, UK in June 1993. The purpose of that meeting was to agree on a standard specification for the Oberon-2 Language, some minimal extensions and a standard portable library. The intention is that all implementors should offer support for Oberon-2 to at least the ETH specification standard and also offer an implementation of the basic library modules. The aim is to ensure that Oberon-2 programs using the library will be consistent and portable across all conforming implementations.

The initial motivation behind the Oakwood meeting was to avoid the fate of Modula-2 being repeated with commercial implementations of Oberon-2. Unfortunately Modula-2 implementations introduced many dialects of the language and many incompatible basic libraries. The standardisation process for Modula-2 took far too long and opened the door to a pandoras box of extensions. The objective of this report is to acknowledge the ETH Oberon-2 language report as the base standard and to provide information useful for compiler developers so that compilers and their basic libraries provide a basic level of compatibility.

1.2 Oberon-2 Language Standard

The standard specification of the language ETH Oberon-2 is contained in a report which is controlled and published by ETH Zrich. The current version of the report is available by anonymous FTP transfer over INTERNET from the directory :

neptune.inf.ethz.ch:~ ftp/Oberon/Docu

The latest complete version of the ETH Oberon-2 report is in file

Oberon2.Report.ps.Z.

A chronological list of all changes made to the report is in file

Oberon2.ChangeList.ps.z

1.3 Use of the name Oberon

The name Oberon has been trademarked by ETH in the context of the operating system and the language. In order to respect the ETH trademark any compiler that does not at least implement ETH Oberon or Oberon-2 should not be referred to or named as an Oberon or Oberon-2 compiler.

When referring to features of ETH Oberon in documentation it is acceptable to use the terms Oberon or Oberon-2. However when referring to any compiler specific extensions the term Oberon should be qualified with an adjective.

For example : “XYZ Oberon-2 supports complex numbers”

In the interest of users, it is strongly recommended that whenever implementors provide a description of their product they specify the extensions that they do or do not support, and the additional libraries they provide.

Implementors should state as part of the description of their compilers whether or not extensions are supported in accordance with these Oakwood Guidelines.

2.0 Language Clarifications

2.1 Introduction

This chapter consists of a list of language clarifications. Ideally there would be no necessity for clarifications and it is hoped that, where relevant, the ETH Oberon-2 Report will be modified at some time in the future. The clarifications listed here are a snapshot of the situation in September ‘93.

2.2 Status of NIL

NIL is a reserved word denoting a predefined value. In contrast to TRUE and FALSE the type of NIL cannot be expressed in Oberon-2.

2.3 Illegal Operations

The following operations are illegal. Their effect is system dependent.

1. De-referencing a NIL pointer.
2. Calling procedure variables with a value NIL.
3. Type tests and type guards with NIL pointers.
4. Indexing an array with an index that is out of range.
5. Accessing a set element outside the range 0 .. MAX (SET).
6. Applying SHORT (...) to an argument with value not in the range of the result type.
7. Operations on strings, or character arrays containing strings, that are not null terminated.
8. Overflows.

2.4 WITH and guarded variables

It is possible to alter a guarded pointer variable within the scope of a guarding WITH statement, example:

```
TYPE
 T = RECORD END; P = POINTER TO T;
 T1 = RECORD (T) END; P1 = POINTER TO T1;
 T2 = RECORD (T) END; P2 = POINTER TO T2;

PROCEDURE X;
 VAR p: P; p1: P1; p2: P2;
 PROCEDURE Y;
 BEGIN
```

```
 p := p2
 END Y;
BEGIN
 NEW (p); NEW(p1); NEW(p2); p := p1;
 WITH p: P1 DO
 Y (*p is now of type P2 and not P1*)
 END
END X;
```

A practical way to handle this is :

If the compiler can be sure it is safe then give no warning message. If there can be any doubt then do give a warning message. A sophisticated compiler could automatically insert the additional relevant type guard checks.

2.5 String Comparison

Strings are always null terminated. Character arrays that are to be compared or used as the source operand of the COPY procedure must contain 0X as a terminator.

The comparison a relop b, where a and b are (open) character arrays or strings and relop is =, #, >, >=, <, <= is performed according to the following pseudocode

```
PROCEDURE Compare (a, b: ARRAY OF CHAR;  relop:RELATION):
BOOLEAN;
 i := 0;
 WHILE (a[i] 1 0X) & (a[i]=b[i])
 DO
 INC (i)
 END;
 RETURN a[i] relop b[i]
END Compare
```

2.6 Recursive declarations and imports

Declarations

The declaration of structured type cannot contain itself. For example a RECORD declaration cannot have itself as the type of one of its fields.

A module must not import itself, for example

```
MODULE x;
 IMPORT x;
END x.
```

However the module name can be used for aliasing, for example

```
MODULE x;
 IMPORT x:=y;
 VAR i: x.INTEGER;
END x.
```

This is, however, bad programming style.

2.7 String and Character Compatibility

A string of length 1 can be used in any context where a character constant is allowed and vice versa.

2.8 Redeclaration of predeclared identifiers

Any predeclared identifier can be redeclared. For example

```
TYPE INTEGER = LONGINT;
```

and

```
PROCEDURE ABS;
BEGIN
 ...
END ABS;
```

Obviously such practice should be discouraged and if used at all used with extreme care.

2.9 Truncation of precision

The type inclusion hierarchy may infer an implicit truncation of precision between REAL and LONGINT. For example, if both types are represented in 32 bits then the REAL mantissa precision is likely to be only 24 bits. An assignment from a LONGINT to a REAL will therefore involve a truncation of precision of value assigned.

3.0 Language Extensions

3.1 Introduction

Language extensions are features provided by compiler developers which are in addition to the language as specified in the ETH Oberon-2 Report.

The purpose of this chapter is not to encourage extensions. The reason for defining them here is to promote a uniform approach to the specification and provision of extensions across different compilers and endeavour to make sure that when the same extension is supported by more than one compiler it has the same syntax and semantics in each. If a particular compiler offers a means to optionally support language extensions then the default compilation option is for no extensions to be enabled.

3.2 Additional Datatypes

Extending ETH Oberon-2 with new data types is a very contentious issue. At the Oakwood meeting the general feeling was that only the complex number type should be considered. Enumerations and unsigned types have been specifically rejected by ETH although they are still found desirable by applications programmers. Unsigned types are particularly important when interfacing to existing external standard libraries such as X Windows, ‘C’ or Windows and had support from the applications programmers. Bit level types are considered to be unnecessary as the SET type can be used.

3.2.1 Type inclusion Hierarchies

Adding data types which are additional to Oberon-2 should be done sympathetically (if at all) and with due consideration to the implications on the whole language. Separate type inclusion hierarchies should be used to separate families of types which are intrinsically incompatible. Explicit conversion procedures should be used to convert values that can be represented in different type inclusion hierarchies. The predefined function procedures LONG and SHORT should provide conversion within any extended type hierarchy.

An example (please note this is NOT a proposal for general implementation)

```
LONGCOMPLEX ⇒ REAL ⇒ LONGINT ⇒ INTEGER ⇒ SHORTINT
LONGCARD ⇒ CARDINAL ⇒ SHORTCARD
LONGCHAR ⇒ CHAR
```

The intention of this scheme is to retain the benefits of type inclusion whilst separating explicitly the system dependent aspects of value conversion between types in different type inclusion hierarchies. Such procedures should be included as built in procedures. For any additional data type extension to the language it is the implementors responsibility to provide an updated version of the Oberon-2 Language Report indicating all the relevant changes required to it.

There is a known problem with this proposal. It does not allow for type inclusion of COMPLEX within LONGREAL. However it was felt to be a better solution than having only one complex type selectable as LONG by compiler switch, which could easily be set to select different options in different modules (and library modules).

<< BK: A proposal is needed for the conversion routine, see Section 3.3.1 >>

3.3 Type COMPLEX and LONGCOMPLEX

Complex numbers are made up of two parts (real, imaginary). The type LONGCOMPLEX is defined as (LONGREAL, LONGREAL), and can be included at the top end of the type inclusion hierarchy. The type COMPLEX is defined as (REAL, REAL) and is an extension of REAL within the hierarchy. See Section 3.4.1. If a value of a type less than or equal to REAL is interpreted as a COMPLEX value then it is considered to be the real part; the corresponding imaginary part is 0. All expression and assignment compatibility rules can be applied to the complex types, for example

```
VAR
 c: COMPLEX;
 r: REAL;
 i: INTEGER;
 c:=i+r;
 c:=c*r;
```

3.3.1 New conversion functions

The following predeclared function procedures are defined, (z) stands for an expression

Name	Argument Type	Result Type	Function
RE(z)	COMPLEX	REAL	Real part
RE(z)	LONGCOMPLEX	LONGREAL	Real part
IM(z)	COMPLEX	REAL	Imaginary part
IM(z)	LONGCOMPLEX	LONGREAL	Imaginary part

<< BK/HM/AF: Predeclared functions SHORT, LONG, MIN, MAX and SIZE need to be defined for COMPLEX and LONGCOMPLEX. >>

3.3.2 Complex literal number syntax

A common notation is used for complex number literals:

number = integer | real | complex.

complex = real “i”.

Examples

```
 Values
 RE IM
1.i 0. 1.
2.+3.i 2. 3.
4. 4. 0.
5.3-6.2i 5.3-6.2
```

3.3.3 Reasons against introducing COMPLEX

The omission of COMPLEX data types from Oberon was a deliberate ETH design decision and not an oversight. The following reasons are cited by Josef Templ.

3.3.3.1 Internal Representation

Cartesian or polar ? Both have advantages, cartesian is more common, though.

3.3.3.2 Efficiency

Utmost efficiency can only be gained by coding COMPLEX operations as REAL operations, because often the real or imaginary parts are zero, one, or a value which allows algebraic simplifications.

3.3.3.3 Accuracy

Not under full programmer control in case of COMPLEX.

3.3.3.4 Difficulties in the hierarchy of numeric types.

The linear type inclusion would be changed to a directed acyclic graph (DAG) if two types COMPLEX and LONGCOMPLEX are introduced with compatibility rules as naturally expected.

A simplification would be to set COMPLEX = LONGCOMPLEX, but is it sufficient ? Another simplification would be to form a separate hierarchy consisting of COMPLEX and LONGCOMPLEX, but is this convenient ?

One should also observe the effect on the rest of the language definition. For example, what about the comparison operators for numeric types ?

3.3.3.5 Implementation

Not the most important point, but COMPLEX also makes the compiler more complex, especially when good code should be generated. The reason is that two separate operand descriptors must be maintained in the compiler to represent the two parts of one complex.

3.3.3.6 Hardware

Unlike INTEGER and REAL, no hardware support for COMPLEX is available.

3.3.3.7 Syntax

Additional syntax is necessary for denoting complex constants and/or additional predeclared functions are necessary.

3.3.3.8 Structured function returns

A common misbelief is that introducing structured function returns would eliminate the discussion about COMPLEX, because then one could define complex operations as functions. It should be noted that this is only half the way since the mathematicians still want to have infix notation which would require the introduction of a more general overloading concept including infix operators. This in turn would break the idea of always qualifying imported objects by the module name.

3.3.3.9 Unused

For the reasons outlined above (efficiency, accuracy), many Fortran programmers don't use complex operations although they are supported by the language.

3.3.3.10 Not sufficient

For the purpose of scientific computing, COMPLEX is only a small step. What is still missing are vector operations and subarrays.

3.4 Interrupt and Code Procedures

A consistent means of providing a clean Oberon-2 interface to highly system dependent features is defined by encapsulating such features in procedures, which are inherently unsafe.

Interrupts are implemented by marking the procedure with + as a prefix

```
PROCEDURE +Proc ... ; ... END Proc;
```

At run-time the procedure has to be associated with the required interrupt using an installation mechanism such as

```
Install (Proc, number);
```

Where number represents a position in a vector table or an actual vector address location, clearly this is implementation specific.

Code procedures are implemented by marking the procedure with a - as a prefix.

```
PROCEDURE -ProcHeading byte {“,”byte};
```

For example

```
PROCEDURE -Sigblock* (mask: SET): SET;
82H, 0, 20H, 109, 91H, 0D0H, 20H, 0;
```

<< BK: A more readable alternative proposed by Steve Metzeler follows, personally I much prefer it. A decision is needed. It would add two new keywords. >>

Interrupts are implemented by marking the procedure with the keyword INTERRUPT as a prefix

```
INTERRUPT PROCEDURE Proc ...; ... END Proc;
```

Similarly code procedures are implemented by marking the procedure with the keyword CODE and giving it a body containing hex byte codes or assembler level instructions.

```
CODE PROCEDURE Proc;
BEGIN
 byte {“,” byte} | {Assembler Instructions}
END Proc;
```

3.5 Interfacing to External Libraries

When Oberon-2 programs are written for external operating systems other than the Oberon System then a mechanism is required to provide interface between them as seamlessly as possible. To avoid performance reduction a direct mapping between Oberon-2 structures and conventions and the external ones is highly desirable. It is also desirable that the notation used should be practical both for large libraries and for individual procedures within a module. It is recognised that use of an external interfacing mechanism renders the module unsafe.

It is recommended that for the benefit of students and newcomers to the language, the documentation of the mechanism bear a health warning in a standard form such as (** NOT SAFE**)

The four elements that must be accommodated for interfacing to external libraries are

- the Oberon-2 name for the facility
- the Oberon-2 type and signature of the facility
- the external name of the facility
- the location name and calling convention style of the library or object.

The following proposal has not been fully tried out however it is offered as a basis for discussion.

For modules containing many procedures all belonging to a single library then the syntax could be

```
MODULE OberonModuleName “[“ convention “]” EXTERNAL “[“
externalLibraryName “]” ...
```

Where convention might be “PASCAL” or “C” and externalLibraryName is also a quoted string.

For example:

```
MODULE ISOStrings [ “Modula-2” ] EXTERNAL
[ “server_XP/lib” ] ...
```

Normal Oberon-2 identifiers within the module are optionally followed by an equivalent external name as a string, for example:

```
PROCEDURE CreateWindow [“CREATE_WINDOW$BIG”] (... );
```

Note that the external non-Oberon-2 identifiers or strings may contain any characters which are valid for the external library.

For an Oberon-2 module that contains just one or a few interface procedures, or is hiding the structure of a set of external modules, then the following form can be used.

```
PROCEDURE “[“<convention>”,”<external library name>”]” ...
```

For example:

```
PROCEDURE [“C”,”Motif.lib”] CreateWindow
 [“CREATE_WINDOW”] (... );
```

<< BK: Please note that Josef Templ and Prof Mössenböck of ETH are strongly against sections 3.6 and 3.7 being suggested as language extensions. >>

3.6 Underscores in Identifiers

Identifiers may contain the additional character “_”

ident = (letter | “_”) {letter | digit | “_”}.

This syntax allows for identifiers to begin with the underscore character “_”.

3.7 In-line Exponentiation

The exponentiation operator “**” provides a convenient notation for arithmetic expressions, rather than using function calls. It is an arithmetic operator which has a higher precedence than the multiply and divide operators. In the expression

```
a := b**c ;
```

value of the result is the value of b raised to the power of the value of c.

<< BK: This introduces a fifth level of precedence into the language. If we include this at all then the full expression grammar needs to be defined so that it can be implemented consistently. Any volunteers please ... >>

4.0 Compilation Control

4.1 Introduction

There are two main issues regarding control of the compilation process, setting of compilation options for the compiler and selection of the specific source text to be compiled. There are also two main schools of thought about how this control should be specified. Application programmers and project managers often like to have a single source text, especially when a program is designed to have many variants (for example a compiler with very similar code generators for a family of processors). Others prefer to use preprocessors to extract the source text of a particular variant first and then compile it. The trend in the market is to integrate preprocessors into compilers, the main reasons being

- readability of the program for maintainers, being able to see the relations between variants.
- direct correlation between compiler error messages and the original source text
- reading the source text only once during the overall compilation process (for speed)
- saving storage (no intermediate versions)

This appears to be an emotive issue with different organisations having strong loyalty to their own particular approach. In this chapter some conventions are defined for control of compilation with the intention that compiler producers offer such features based on the same basic model. It is recognised that the choice of notation may be prescribed by the operating system in use or to fall in line with the conventions used on existing compilers. Even so where there are opportunities to follow the guidelines and to reduce variation they should be taken.

The additional language constructs defined below should not be considered to be part of the Oberon-2 language. Rather they define a separate compiler control language that coexists with and is distinct from the Oberon-2 language.

All in-line commands to the compiler are contained in ISO style pseudo comments using angled brackets <* ... *>.

4.2 Runtime checks

Runtime checks are controlled by pragmas which are used to selectively enable and disable each option. All pragmas should default to provide maximum safety.

Syntax : “<*$” {modifier} “*>”

where modifier is

- pragma -set pragma OFF, disable
- pragma +set pragma ON, enable

- < stack the current pragma state
- > unstack the current pragma state
- ! revert to the pragma state defined by the original command line.

The following letters are from the ETH OP2 compiler and are only shown as a guide. In practice they are likely to be implementation specific for other compatibility reasons (e.g. other compilers, Unix ...)

pragma	**default**	**meaning**
A	+	ASSERT generation
K	+	Stack overflow check
P	+	Pointer initialisation
R	+	Range check (e.g. SHORT (Int) is in the SHORTINT range)
S	-	Allow symbol file to replace the previous version if it differs
T	+	Type check (suppress type guards)
V	+	Overflow check
X	+	Index check, both static and dynamic

Source pragmas can be either upper or lower case.

Note : The ETH compilers have a default of - for the R and V pragmas.

4.3 Compiler option control

Compiler options can be turned on and off using the statements. As they apply to a whole compilation unit it only makes sense to use them at the beginning of a module.

<*OPTION+ *> to set OPTION on, enabling it
<*OPTION- *> to set OPTION off, disabling it

For example <*STANDARD+ *>

The options are

Option	**default**	**meaning**
STANDARD	+	Oberon-2 Report standard, no extensions allowed
INITIALISE	+	All pointers are initialised
MAIN	+	Generates a program entry point. Only one per system !
WARNINGS	+	Report questionable usage

Note: There is no option for controlling garbage collection, for example for systems which need deterministic timing. This can be achieved by explicitly calling the system memory manager to turn garbage collection off and on. Also see 5.6.

4.4 Compiler source control

For large programs where a single source text must support many runtime variants there is a practical need for selective compilation of the source text. The selection can be made either using a preprocessor or, for reasons of optimising disk storage, speed and efficiency, at compiler time.

The syntax for expressing the source text selection is

```
<* IF condition THEN *>
<* ELSIF condition THEN *>
<* ELSE *>
<* END *>
```

The conditional expression consists of programmer defined SELECTORS which can be combined as an Oberon-like boolean expression which can contain the operators ~, &, OR. Compiler options are in effect predefined selectors and can be used within the condition part

To define a new SELECTOR, which has a default value of FALSE

```
<* NEW SelectorName *>
```

To give a SELECTOR a value

```
<* SelectorName+ *> to set it TRUE
<* SelectorName- *> to set it FALSE
```

Examples:

```
<*IF ~ MAIN THEN *> ...

<*IF M68000 & WARNINGS THEN *>
 IMPORT CG68000;
<*ELSE *>
 IMPORT CG80x86;
<*END*>
```

5.0 Implementation Recommendations

5.1 Introduction

This chapter includes recommendations describing some specific characteristics for compilers which confom to these guidelines.

5.2 Type ranges

The minimum value that is returned by MAX (type) and the maximum value returned by MIN (type) should be at least (at most) as follows

TYPE	**'MAX' VALUE**	**'MIN' VALUE**
SHORTINT	127	-128
INTEGER	32767	-32768
LONGINT	+2147483647	-2147483648
REAL	IEEE 32 bit format if possible	
LONGREAL	at least the precision of REAL IEEE format, higher resolution if possible	
SET	32 elements minimum (0..31)	
CHAR	0..0FFX where ...	
	00..7FX ASCII code	
	80 ..0FFX ISO LATIN-1 CODE preferred, but code set not defined	

5.3 Type Extension Levels

If an implementation imposes a limit on the number of levels of type extension it should not be less than 8 levels including the base type.

5.4 The module SYSTEM

The module SYSTEM should be based on the ETH model wherever reasonable.

5.5 The procedure SYSTEM.MOVE

For the procedure SYSTEM.MOVE the behaviour when the source extent and destination extent overlap should be made clear regarding overwriting, also the special case when length = 0.

5.6 Garbage collection

Automatic garbage collection is recommended wherever possible. If garbage collection is not available or a mechanism is available to activate and deactivate it then a procedure

DISPOSE can be provided in the module system. It takes a single parameter which is a pointer value parameter.

5.7 Implementation characteristics

Each compiler implementation inevitably has limits, for example to identifier length or runtime checks provided. A list of characteristics may be provided for each implementation so that users can judge its suitability and any portability problems that might arise when moving between implementations.

The following characteristics are defined

Length of identifier, at least 23 significant characters possible

Record extension levels, 8 including base type

Actual type sizes (INTEGER, LONGINT, ...), see 5.2

5.8 Initialisation of Pointers

All pointers for procedure variables, variables, record fields and array elements should be initialised by the compiler to a safe value, the value NIL is recommended. This applies to pointers which are statically allocated, dynamically allocated or on stacks. Refer to the ETH change list. (Section 1.2).

The ETH Report does not define that variables are initialised however a practical implementation might include the following approach ...

The compiler should guarantee that level 0 variables of any pointer or procedure type are either statically or dynamically initialised to NIL before the initialisation part of a module (module body) is executed.

The compiler should provide code to dynamically initialise local variables of any pointer or procedure type to NIL before the procedure body is executed.

When executing the predeclared procedure NEW, the storage/heap manager of the run-time system (if any) should initialise the heap space to NIL. Alternatively the compiler should emit code to initialise dynamic variables of any pointer or procedure type to NIL.

A compilation switch may be provided to inhibit the generation of initialisation code for variables of pointer or procedure types. In case of dynamic variables allocated with the procedure NEW, an alternative storage (or run-time system) module may be provided which does no initialisation.

5.9 Handling undefined semanics

When operations with undefined semantics, as listed in Section 2.3, occur then their effect is system-dependent and should be handled in consistent ways within a particular implementation. It is expected that the program would terminate with a message indicating the cause and its program location.

5.10 Monadic ‘-’

It should be made clear in documentation supplied with compilers that monadic negation is an addition operator and has a lower precedence then the multiplication operator. For example the expression -5 MOD 3 is equivalent to -(5 MOD 3).

5.11 Conversion from Integer to Real

It should be made clear to compiler users that the function LONG cannot be used to convert an expression of LONGINT type to REAL type. There is no explicit function for that purpose. An assignment of the form

real := integer;

has to be used which automatically converts from any integer type to REAL type.

5.12 Exported Comments

An exported comment is denoted using two consecutive asterisks after the opening bracket, for example

(** this is an exported comment *)

It signals to a browser that the comment should be included in a DEFINITION module being derived from the module being processed. It is a convention rather than a language issue.

5.13 Read only VAR Parameters

There have been many requests to make ARRAY and RECORD parameters read-only to achieve the efficiency of passing by reference without the associated possibility for corruption of the calling parameter. An attempt to make an assignment to any component of such a read only parameter is a compile-time error. Such parameters could be marked with the standard read only “-” symbol. For example:

```
PROCEDURE Print (theText-: ARRAY OF CHAR) ;
```

Discussions with ETH suggest this is really a compiler code optimisation issue and on this basis it is recommended that this extension is not implemented.

5.14 Type Guards with RECORD parameters

If a record is assigned to a formal VAR parameter record, the compiler must generate an implicit type test to make sure that the static type and the dynamic type of the destination record are the same.

6.0 Library Modules

6.1 Introduction

It is very desirable for programmers that a basic set of library modules is available across a range of different compiler implementations. On the other hand it is also clear that it is impossible to design library modules that are useful for all purposes. To be effective library modules must have a purpose which makes sense for the library user.

This report defines two groups of modules

- modules based on the ETH Oberon System designs which provide input-output facilities and support for published teaching material, in particular the series of Oberon books from ETH authors
- modules which extend the functionality of the language in a standardised way, for example maths libraries.

The module definitions provided in Appendix 1 are intended to encourage all compiler developers to offer sets of library modules with the same interface and functionality.

All implementations should support all the so called basic modules described in Section 6.2. The additional modules should be provided if they are relevant to the particular compiler implementation (e.g. if COMPLEX is supported).

6.2 Basic Modules

It is intended that the basic modules are provided with all Oberon-2 compiler implementations. They are based on ETH Oberon System designs ...

- XYplane Elementary pixel plotting
- Input Keyboard and pointer device access
- In Inputting from a standard stream
- Out Outputting to a standard stream
- Files File input output, with riders
- Strings Simple manipulation for strings
- Math Math and trig functions for REAL
- MathL Math and trig functions for LONGREAL

6.3 Additional Modules

The additional modules are provided with compiler implementations on an ‘as needed’ basis ...

- Coroutines — Provides non-preemptive threads each with its own stack but all sharing a common address space.
- MathC — Maths functions for COMPLEX
- MathLC — Maths functions for LONGCOMPLEX

Appendix A: Library modules

1.1 Basic Library Modules

It is expected that all Oberon-2 compiler implementations will include the following modules ...

XYplane Input In Out Files Strings Math and MathL

1.2 Additional Modules

The following modules are optional. If they are provided then they should follow the specifications ...

- Coroutines
- MathC and MathLC

1.2.1 Module XYplane

Module XYplane provides some basic facilities for graphics programming. Its interface is kept as simple as possible and is therefore more suited for programming exercises than for serious graphics applications.

XYplane provides a Cartesian plane of pixels that can be drawn and erased. The plane is mapped to some location on the screen. The variables X and Y indicate its lower left corner, W its width and H its height. All variables are read-only.

```
DEFINITION XYplane;
CONST draw = 1; erase = 0;
VAR X, Y, W, H: INTEGER;
PROCEDURE Open;
PROCEDURE Clear;
PROCEDURE Dot (x, y, mode: INTEGER);
PROCEDURE IsDot (x, y: INTEGER): BOOLEAN;
PROCEDURE Key (): CHAR;
END XYplane.
```

1.2.1.1 Operations

Open initializes the drawing plane.

Clear erases all pixels in the drawing plane.

Dot(x, y, m) draws or erases the pixel at the coordinates (x, y) relative to the lower left corner of the plane. If m=draw the pixel is drawn, if m=erase the pixel is erased.

IsDot(x, y) returns TRUE if the pixel at the coordinates (x, y) relative to the lower left corner of the screen is drawn, otherwise it returns FALSE.

Key() reads the keyboard. If a key was pressed prior to invocation, its character value is returned, otherwise the result is 0X.

1.2.1.2 Remarks

In the ETH Oberon System Open opens a viewer that takes the whole user track. The contents of this viewer is the drawing plane provided by XYplane.

1.2.1.3 Origin

Designed by Martin Reiser for the book ‘Programming in Oberon’. The above specification was proposed by H Mössenböck, ETH

1.2.2 Module Input

Module Input provides facilities to access the mouse, the keyboard, and the clock.

```
DEFINITION Input;
 VAR TimeUnit: LONGINT;
PROCEDURE Available (): INTEGER;
PROCEDURE Read (VAR ch: CHAR);
PROCEDURE Mouse (VAR keys: SET; VAR x, y: INTEGER);
PROCEDURE SetMouseLimits (w, h: INTEGER);
PROCEDURE Time (): LONGINT;
 END Input.
```

1.2.2.1 State

Keyboard buffer. A queue of characters typed in from the keyboard.

Time. Elapsed time since system startup in units of size 1/TimeUnit seconds.

1.2.2.2 Operations

Available() returns the number of characters in the keyboard buffer.

Read(ch) returns (and removes) the next character from the keyboad buffer. If the buffer is empty, Read waits until a key is pressed.

Mouse(keys, x, y) returns the current mouse position (x, y) in pixels relative to the lower left corner of the screen. keys is the set of the currently pressed mouse keys (left = 2, middle = 1, right = 0).

SetMouseLimits(w, h) defines the rectangle where the mouse moves (in pixels). Subsequent calls to the operation Mouse will return coordinates for x in the range 0..w-1 and y in the range 0..h-1.

Time() returns the time elapsed since system startup in units of size 1/TimeUnit seconds.

1.2.2.3 Examples

```
IF Input.Available() > 0 THEN Input.Read(ch) END;
 REPEAT
 Input.Mouse(keys, x, y);
 ... draw mouse cursor at position (x, y) ...
UNTIL keys = {}
seconds := Input.Time() DIV Input.TimeUnit
```

1.2.2.4 Origin

Part of the ETH Oberon System. The above specification was proposed by H Mössenböck, ETH.

1.2.3 Module In

Module In provides a set of basic routines for formatted input of characters, character sequences, numbers, and names. It assumes a standard input stream with a current position that can be reset to the beginning of the stream.

```
DEFINITION In;
  VAR Done: BOOLEAN;
 PROCEDURE Open;
  PROCEDURE Char (VAR ch: CHAR);
  PROCEDURE Int (VAR i: INTEGER);
  PROCEDURE LongInt (VAR i: LONGINT);
  PROCEDURE Real (VAR x: REAL);
  PROCEDURE LongReal (VAR y: LONGREAL);
  PROCEDURE String (VAR str: ARRAY OF CHAR);
  PROCEDURE Name (VAR name: ARRAY OF CHAR);
 END In.
```

1.2.3.1 State

Current position. The character position in the input stream from where the next symbol is read. Open (re)sets it to the beginning of the input stream. After reading a symbol the current position is set to the position immediately after this symbol. Before the first call to Open the current position is undefined.

Done. Indicates the success of an input operation. If Done is TRUE after an input operation, the operation was successful and its result is valid. An unsuccessful input operation sets Done to FALSE; it remains FALSE until the next call to Open. In particular, Done is set to FALSE if an attempt is made to read beyond the end of the input stream.

1.2.3.2 Operations

Open (re)sets the current position to the beginning of the input stream. Done indicates if the operation was successful.

The following operations require Done = TRUE and guarantee (Done = TRUE and the result is valid) or (Done = FALSE). All operations except Char skip leading blanks, tabs or end-of-line characters.

Char(ch) returns the character ch at the current position.

LongInt(n) and Int(n) return the (long) integer constant n at the current position according to the format:

IntConst = digit {digit} | digit {hexDigit} “H”.

Real(n) returns the real constant n at the current position according to the format:

RealConst = digit {digit} [{digit} [“E” (“+” | “-”)

digit {digit}]].

LongReal(n) returns the long real constant n at the current position according to the format:

LongRealConst = digit {digit} [{digit} [(“D” |”E”)

(“+” | “-”) digit {digit}]].

String(s) returns the string s at the current position according to the format:

StringConst = ‘”’ char {char} ‘”’.

The string must not contain characters less than blank such as EOL or TAB.

Name(s) returns the name s at the current position according to the file name format of the underlying operating system (e.g. “lib/My.Mod” under Unix)

Example:

```
  VAR i: INTEGER; ch: CHAR; r: REAL; s, n: ARRAY 32 OF CHAR;
...
In.Open;
In.Int(i); In.Char(ch); In.Real(r); InString(s);In.Name(n)
```

Input stream:

```
  123*1.5 “abc” Mod.Proc
```

Results:

```
  i = 123
 ch = “*”
 r = 1.5E0
 s = “abc”
 n = “Mod.Proc”
```

1.2.3.3 Remarks

In the ETH Oberon System the input stream is the text immediately following the most recently invoked command. If this text starts with the character “^” the current position is set to the beginning of the most recent selection (if no selection exists, Done = FALSE). If the text starts with the character “*” the current position is set to the beginning of the text in the marked viewer (if no viewer is marked, Done = FALSE). The end of the input stream is the end of the text containing the current position. There is no provision for input of SHORT integers.

1.2.3.4 Origin

Designed by Martin Reiser for the book ‘Programming in Oberon’. The above specification was proposed by H Mössenböck, ETH.

1.2.4 Module Out

Module Out provides a set of basic routines for formatted output of characters, numbers, and strings. It assumes a standard output stream to which the symbols are written.

```
DEFINITION Out;
 PROCEDURE Open;
 PROCEDURE Char (ch: CHAR);
 PROCEDURE String (str: ARRAY OF CHAR);
 PROCEDURE Int (i, n: LONGINT);
 PROCEDURE Real (x: REAL; n: INTEGER);
 PROCEDURE LongReal (x: LONGREAL; n: INTEGER);
 PROCEDURE Ln;
END Out.
```

1.2.4.1 Operations

Open initializes the output stream.

Char(ch) writes the character ch to the end of the output stream

String(s) writes the null-terminated character sequence s to the end of the output stream (without 0X).

Int(i, n) writes the integer number i to the end of the output stream. If the textual representation of i requires m characters, i is right adjusted in a field of Max(n, m) characters padded with blanks at the left end. A plus sign is not written.

Real(x, n) writes the real number x to the end of the output stream using an exponential form. If the textual representation of x requires m characters (including a two-digit signed exponent), x is right adjusted in a field of Max(n, m) characters padded with blanks at the left end. A plus sign of the mantissa is not written.

LongReal(x, n) writes the long real number x to the end of the output stream using an exponential form. If the textual representation of x requires m characters (including a three-digit signed exponent), x is right adjusted in a field of Max(n, m) characters padded with blanks at the left end. A plus sign of the mantissa is not written.

Ln writes an end-of-line symbol to the end of the output stream.

Examples

	output (asterisks denote blanks)
	`Out.Open;`
`Out.Int(-3, 5);`	`***3`
`Out.Int(3, 0);`	`3`
`Out.Real(1.5, 10);`	`**1.50E+00`
`Out.Real(-0.005, 0)`	`-5.0E-03`

1.2.4.2 Remarks

In the ETH Oberon System the output is appended to an output text that is cleared when module Out is loaded. The output text can be displayed in a new viewer by a call to the procedure Open (Open can also be called as a command).

1.2.4.3 Origin

Designed by Martin Reiser for the book ‘Programming in Oberon’. The above specification was proposed by H Mössenböck, ETH.

1.2.5 Module Files

Module Files provides operations on files and the file directory.

```
DEFINITION Files;
  IMPORT SYSTEM;
```

```
TYPE
File = POINTER TO Handle;
Rider = RECORD
  eof: BOOLEAN;
  res: LONGINT;
END;
PROCEDURE Old (name: ARRAY OF CHAR): File;
PROCEDURE New (name: ARRAY OF CHAR): File;
PROCEDURE Register (f: File);
PROCEDURE Close (f: File);
PROCEDURE Purge (f: File);
PROCEDURE Delete (name: ARRAY OF CHAR; VAR res: INTEGER);
PROCEDURE Rename (old, new: ARRAY OF CHAR;
                  VAR res: INTEGER);
PROCEDURE Length (f: File): LONGINT;
PROCEDURE GetDate (f: File; VAR t, d: LONGINT);
PROCEDURE Set (VAR r: Rider; f: File; pos: LONGINT);
PROCEDURE Pos (VAR r: Rider): LONGINT;
PROCEDURE Base (VAR r: Rider): File;
PROCEDURE Read (VAR r: Rider; VAR x: SYSTEM.BYTE);
PROCEDURE ReadInt (VAR R: Rider; VAR x: INTEGER);
PROCEDURE ReadLInt (VAR R: Rider; VAR x: LONGINT);
PROCEDURE ReadReal (VAR R: Rider; VAR x: REAL);
PROCEDURE ReadLReal (VAR R: Rider; VAR x: LONGREAL);
PROCEDURE ReadNum (VAR R: Rider; VAR x: LONGINT);
PROCEDURE ReadString (VAR R: Rider; VAR x: ARRAY OF CHAR);
PROCEDURE ReadSet (VAR R: Rider; VAR x: SET);
PROCEDURE ReadBool (VAR R: Rider; VAR x: BOOLEAN;
PROCEDURE ReadBytes (VAR r: Rider;
                     VAR x: ARRAY OF SYSTEM.BYTE;
                     n: LONGINT);
PROCEDURE Write (VAR r: Rider; x: SYSTEM.BYTE);
PROCEDURE WriteInt (VAR R: Rider; x: INTEGER);
PROCEDURE WriteLInt (VAR R: Rider; x: LONGINT);
PROCEDURE WriteReal (VAR R: Rider; x: REAL);
PROCEDURE WriteLReal (VAR R: Rider; x: LONGREAL);
PROCEDURE WriteNum (VAR R: Rider; x: LONGINT);
PROCEDURE WriteString (VAR R: Rider; x: ARRAY OF CHAR);
PROCEDURE WriteSet (VAR R: Rider; x: SET);
PROCEDURE WriteBool (VAR R: Rider; x: BOOLEAN);
```

```
PROCEDURE WriteBytes (VAR r: Rider;
                      VAR x: ARRAY OF SYSTEM.BYTE;
                      n: LONGINT)
END Files.
```

1.2.5.1 Types

A File represents a stream of bytes usually stored on an external medium. It has a certain length as well as the date and time of its last modification.

A file directory is a mapping from file names to files. A file that is not registered in the directory is considered temporary.

A Rider is a read/write position in a file (positions start with 0). There may be multiple riders set to the same file. The field eof is set to TRUE if an attempt was made to read beyond the end of the file. The field res reports the success of ReadBytes and WriteBytes operations. Writing data overwrites old data at the rider position. When data is written beyond the end of the file, the file length increases.

1.2.5.2 Operations on files and the file directory

Old(fn) searches the name fn in the directory and returns the corresponding file. If the name is not found, it returns NIL.

New(fn) creates and returns a new file. The name fn is remembered for the later use of the operation Register. The file is only entered into the directory when Register is called.

Register(f) enters the file f into the directory together with the name provided in the operation New that created f. The file buffers are written back. Any existing mapping of this name to another file is overwritten.

Close(f) writes back the file buffers of f. The file is still accessible by its handle f and the riders positioned on it. If a file is not modified it is not necessary to close it.

Purge(f) resets the length of file f to 0.

Delete(fn, res) removes the directory entry for the file fn without deleting the file. If res=0 the file has been successfully deleted. If there are variables referring to the file while Delete is called, they can still be used.

Rename(oldfn, newfn, res) renames the directory entry oldfn to newfn. If res=0 the file has been successfully renamed. If there are variables referring to the file while Rename is called, they can still be used.

Length(f) returns the number of bytes in file f.

GetDate(f, t, d) returns the time t and date d of the last modification of file f. The encoding is: hour = t DIV 4096; minute = t DIV 64 MOD 64; second = t MOD 64; year = d DIV 512; month = d DIV 32 MOD 16; day = d MOD 32.

1.2.5.3 Operations on riders

Set(r, f, pos) sets the rider r to position pos in file f. The field r.eof is set to FALSE. The operation requires that 0 <= pos < Length(f).

Pos(r) returns the position of the rider r.

Base(r) returns the file to which the rider r has been set.

1.2.5.4 Operations for unformatted input and output

In general, all operations must use the following format for external representation:

'Little endian' representation (i.e., the least significant byte of a word is the one with the lowest address on the file).

Numbers: SHORTINT 1 byte, INTEGER 2 bytes, LONGINT 4 bytes
Sets: 4 bytes, element 0 is the least significant bit
Booleans: single byte with FALSE = 0, TRUE = 1
Reals: IEEE standard; REAL 4 bytes, LONGREAL 8 bytes
Strings: with terminating 0X

1.2.5.5 Reading

Read(r, x) reads the next byte x from rider r and advances r accordingly.

ReadInt(r, i) and ReadLInt(r, i) read a (long) integer number i from rider r and advance r accordingly.

ReadReal(r, x) and ReadLReal(r, x) read a (long) real number x from rider r and advance r accordingly.

ReadNum(r, i) reads an integer number i from rider r and advances r accordingly. The number i is compactly encoded (see remarks below).

ReadString(r, s) reads a sequence of characters (including the terminating 0X) from rider r and returns it in s. The rider is advanced accordingly. The actual parameter corresponding to s must be long enough to hold the character sequence plus the terminating 0X.

ReadSet(r, s) reads a set s from rider r and advances r accordingly.

ReadBool(r, b) reads a Boolean value b from rider r and advances r accordingly.

ReadBytes(r, buf, n) reads n bytes into buffer buf starting at the rider position r. The rider is advanced accordingly. If less than n bytes could be read, r.res contains the number of requested but unread bytes.

1.2.5.6 Writing

Write(r, x) writes the byte x to rider r and advances r accordingly.

WriteInt(r, i) and WriteLInt(r, i) write the (long) integer number i to rider r and advance r accordingly.

WriteReal(r, x) and WriteLReal(r, x) write the (long) real number x to rider r and advance r accordingly.

WriteString(r, s) writes the sequence of characters s (including the terminating 0X) to rider r and advances r accordingly.

WriteNum(r, i) writes the integer number i to rider r and advances r accordingly. The number i is compactly encoded (see remarks below).

WriteSet(r, s) writes the set s to rider r and advances r accordingly.

WriteBool(r, b) writes the Boolean value b to rider r and advances r accordingly.

WriteBytes(r, buf, n) writes the first n bytes from buf to rider r and advances r accordingly. r.res contains the number of bytes that could not be written (e.g., due to a disk full error).

1.2.5.7 Examples

```
VAR f: Files.File; r: Files.Rider; ch: CHAR;
```

Reading from an existing file xxx:

```
f := Files.Old(“xxx”);
IF f # NIL THEN
 Files.Set(r, f, 0);
 Files.Read(r, ch);
 WHILE ~ r.eof DO ... Files.Read(r, ch) END
END
```

Writing to a new file yyy:

```
f := Files.New(“yyy”);
Files.Set(r, f, 0);
Files.WriteInt(r, 8); Files.WriteString(r, “ bytes”);
Files.Register(f)
```

1.2.5.8 Remarks

WriteNum and ReadNum, should use the following encoding algorithms for conversion to and from external format.

```
PROCEDURE WriteNum (VAR r: Rider; x: LONGINT);
BEGIN
 WHILE (x < - 64) OR (x > 63) DO
 Write(r, CHR(x MOD 128 + 128)); x := x DIV 128
 END;
 Write(r, CHR(x MOD 128))
END WriteNum;

PROCEDURE ReadNum (VAR r: Rider; VAR x: LONGINT);
 VAR s: SHORTINT; ch: CHAR; n: LONGINT;
BEGIN
 s := 0; n := 0;
 Read(r, ch);
 WHILE ORD(ch) >= 128 DO
 INC(n, ASH(ORD(ch) - 128, s) );
 INC(s, 7);
 Read(r, ch)
 END;
 x := n + ASH(ORD(ch) MOD 64 - ORD(ch) DIV 64 * 64, s)
END ReadNum;
```

The reason for the specification of the file name in the operation New is to allow allocation of the file on the correct medium from the beginning (if the operating system supports multiple media).

The operations Read, Write, ReadBytes and WriteBytes require the existence of a type SYSTEM.BYTE with the following characteristics:

If a formal parameter is of type SYSTEM.BYTE the corresponding actual parameter may be of type CHAR, SHORTINT, or SYSTEM.BYTE.

If a formal variable parameter is of type ARRAY OF SYSTEM.BYTE the corresponding actual parameter may be of any type. Note that this feature is dangerous and inherently unportable. Its use should therefore be restricted to system-level modules.

1.2.5.9 Origin

This module is part of the ETH Oberon System. The above specification was proposed by H Mössenböck, ETH.

1.2.6 Module Strings

Module Strings provides a set of operations on strings (i.e., on string constants and character arrays, both of which contain the character 0X as a terminator). All positions in strings start at 0.

```
DEFINITION Strings;
    PROCEDURE Length (s: ARRAY OF CHAR): INTEGER;
    PROCEDURE Insert (source: ARRAY OF CHAR;
                      pos: INTEGER;
                      VAR dest: ARRAY OF CHAR);
    PROCEDURE Append (extra: ARRAY OF CHAR;
                      VAR dest: ARRAY OF CHAR);
    PROCEDURE Delete (VAR s: ARRAY OF CHAR;
                      pos, n: INTEGER);
    PROCEDURE Replace (source: ARRAY OF CHAR;
                      pos: INTEGER;
                      VAR dest: ARRAY OF CHAR);
    PROCEDURE Extract (source: ARRAY OF CHAR;
                      pos, n: INTEGER;
                      VAR dest: ARRAY OF CHAR);
    PROCEDURE Pos    (pattern, s: ARRAY OF CHAR;
                      pos: INTEGER): INTEGER;
    PROCEDURE Cap    (VAR s: ARRAY OF CHAR);
END Strings
```

1.2.6.1 Operations

Length(s) returns the number of characters in s up to and excluding the first 0X.

Insert(src, pos, dst) inserts the string src into the string dst at position pos (0 <=pos<=Length(dst)). If pos = Length(dst),src is appended to dst. If the size of dst is not large enough to hold the result of the operation, the result is truncated so that dst is always terminated with a 0X.

Append(s,dst) has the same effect as Insert(s,Length(dst),dst).

Delete(s, pos, n) deletes n characters from s starting at position pos (0 <= pos Length(s)). If n > Length(s) - pos, the new length of s is pos.

Replace(src, pos, dst) has the same effect as Delete(dst, pos, Length(src)) followed by an Insert(src, pos, dst).

Extract(src, pos, n, dst) extracts a substring dst with n characters from position pos (0 <= pos Length(src)) in src. If n > Length(src) - pos, dst is only the part of src from pos to the

end of src, i.e. Length(src) -1. If the size of dst is not large enough to hold the result of the operation, the result is truncated so that dst is always terminated with a 0X.

Pos(pat, s, pos) returns the position of the first occurrence of pat in s. Searching starts at position pos. If pat is not found, -1 is returned.

Cap(s) replaces each lower case letter within s by its upper case equivalent.

1.2.6.2 Remarks

String assignments and string comparisons are already supported by the language Oberon-2.

1.2.6.3 Origin

This module is loosely based on the ISO Modula-2 Strings library but is much simplified. It was edited by Brian Kirk, Nick Walsh, Josef Templ and Hanspeter Mössenböck.

1.2.7 Module Math and MathL

The module Math provides a basic set of general purpose functions using REAL arithmetic. The module MathL provides the same functions for LONGREAL arithmetic.

```
DEFINITION Math;
 CONST
 pi = 3.14159265358979323846;
 e = 2.71828182845904523536;
 PROCEDURE sqrt (x : REAL) : REAL;
 PROCEDURE power (x,base : REAL) : REAL;
 PROCEDURE exp (x : REAL): REAL;
 PROCEDURE ln (x : REAL) : REAL;
 PROCEDURE log (x,base : REAL) : REAL;
 PROCEDURE round (x : REAL) : REAL;
 PROCEDURE sin (x : REAL) : REAL;
 PROCEDURE cos (x : REAL) : REAL
 PROCEDURE tan (x : REAL) : REAL;
 PROCEDURE arcsin (x : REAL) : REAL;
 PROCEDURE arccos (x : REAL) : REAL;
 PROCEDURE arctan (x : REAL) : REAL;
 PROCEDURE arctan2(x,y : REAL): REAL
 PROCEDURE sinh (x:REAL):REAL;
 PROCEDURE cosh (x:REAL):REAL;
 PROCEDURE tanh (x:REAL):REAL;
 PROCEDURE arcsinh(x:REAL):REAL;
```

```
 PROCEDURE arccosh(x:REAL):REAL;
 PROCEDURE arctanh(x:REAL):REAL;
END Math.
```

1.2.7.1 Operations

sqrt (x)returns the square root of x, where x must be positive

sin, cos, tan (x)returns the sine, cosine or tangent value of x, where x is in radians

arcsin, arcos, arctan (x) returns the arcsine, arcos, arctan value in radians of x, where x is in the sine, cosine or tangent value

power(x, base)returns the x to the power base

round(x) if fraction part of x is in range 0.0 to 0.5 then the result is the largest integer not greater than x, otherwise the result is x rounded up to the next highest whole number. Note that integer values cannot always be exactly represented in REAL or LONGREAL format.

ln(x) returns the natural logarithm (base e) of x

exp(x)is the exponential of x base e. x must not be so small that this exponential underflows nor so large that it overflows.

log(x,base)is the logarithm of x base b. All positive arguments are allowed. The base b must be positive.

arctan2(xn,xd)is the quadrant-correct arc tangent atan(xn/xd). If the denominator xd is zero, then the numerator xn must not be zero. All arguments are legal except xn = xd = 0.

sinh(x) is the hyperbolic sine of x. The argument x must not be so large that exp(|x|) overflows.

cosh(x) is the hyperbolic cosine of x. The argument x must not be so large that exp(|x|) overflows.

tanh(x) is the hyperbolic tangent of x. All arguments are legal.

arcsinh(x) is the arc hyperbolic sine of x. All arguments are legal.

arccosh(x) is the arc hyperbolic cosine of x. All arguments greater than or equal to 1 are legal.

arctanh(x) is the arc hyperbolic tangent of x.

|x| < 1 - sqrt(em), where em is machine epsilon.

Note that |x| must not be so close to 1 that the result is less accurate than half precision.

1.2.7.2 Source:

Based on the original ETH Math module, with additions from BK and Al Freed, NASA.

<< BK should the result of round be LONGINT or LONGREAL ? >>

<< AF round (LONGREAL) will have a precision problem anyway. >>

1.2.8 Module Coroutines

Module Coroutines provides non-preemptive threads each with its own stack but otherwise sharing a common address space. Coroutines can explicitly transfer control to other coroutines which are then resumed from the point where they did their last transfer of control.

```
DEFINITION Coroutines;
 TYPE
 Coroutine = RECORD END;
 Body = PROCEDURE;
 PROCEDURE Init (body: Body; stackSize: LONGINT;
                 VAR cor: Coroutine);
 PROCEDURE Transfer (VAR from, to: Coroutine);
END Coroutines.
```

1.2.8.1 Operations

Init(p, s, c)creates and initialises a new coroutine c with a stack of s bytes and a body provided as the procedure p. An initialised coroutine can be started by a Transfer to it. In this case its execution will start at the first instruction of p. Procedure p must never return.

Transfer(f, t) transfers control from the currently executing coroutine to the coroutine t. The state of the currently executing coroutine is saved in f. When control is transferred back to f later, f will be restarted in the saved state.

1.2.8.2 Source

Proposed by Prof Hanspeter Mössenböck, ETH.

1.2.9 Modules MathC and MathLC

The module MathC provides functions for COMPLEX arithmetic. The module MathLC provides the same functions for LONGCOMPLEX.

```
DEFINITION MathC;
 PROCEDURE abs (z:COMPLEX):REAL;
 PROCEDURE power(z:COMPLEX;base:REAL):COMPLEX;
```

```
 PROCEDURE conj(z:COMPLEX):COMPLEX;
 PROCEDURE sqrt(z:COMPLEX):COMPLEX;
 PROCEDURE exp (z:COMPLEX):COMPLEX;
 PROCEDURE ln (z:COMPLEX):COMPLEX;
 PROCEDURE log (z:COMPLEX, b:REAL):COMPLEX;
 PROCEDURE sin (z:COMPLEX):COMPLEX;
 PROCEDURE cos (z:COMPLEX):COMPLEX;
 PROCEDURE tan (z:COMPLEX):COMPLEX;
 PROCEDURE arcsin (z:COMPLEX):COMPLEX;
 PROCEDURE arccos (z:COMPLEX):COMPLEX;
 PROCEDURE arctan (z:COMPLEX):COMPLEX;
 PROCEDURE arctan2 (zn,zd:COMPLEX):COMPLEX;
 PROCEDURE sinh (z:COMPLEX):COMPLEX;
 PROCEDURE cosh (z:COMPLEX):COMPLEX;
 PROCEDURE tanh (z:COMPLEX):COMPLEX;
 PROCEDURE arcsinh (z:COMPLEX):COMPLEX;
 PROCEDURE arccosh (z:COMPLEX):COMPLEX;
 PROCEDURE arctanh (z:COMPLEX):COMPLEX;
END MathC.
```

1.2.9.1 Operations

z = x + iy

bn	is the biggest floating point number of a given machine.
em	is machine epsilon.
es	is em divided by the machine arithmetic base.
sn	is the smallest floating point number of a given machine.

abs(z)is the absolute value or magnitude of the complex number z. The arguments x and y must not be so large that x*x + y*y overflows. The returned value is a real number.

power (Z,base) returns Z to the power base, see comments for exp.

conj(z) is the complex conjugate of z. All arguments are legal.

sqrt(z) is the complex square root of z. The absolute value of z must not overflow.

exp(z) is the complex exponential of z to the base e. The real part of z, i.e. x, should not be so small that the result underflows nor so large that it overflows. If |y| is too large, the result may be less accurate than half precision. If |y| is extremely large, the result will have no precision.

ln(z) is the complex natural logarithm (base e) of z. The argument must not be zero, and the absolute value of z must not overflow.

log(z,b) is the complex natural logarithm of z to the base b. The argument must not be zero, and the absolute value of z must not overflow. Base b must be positive.

sin(z) is the complex sine of z.

|Re(z)| = |x| <= 1/sqrt(em) = x(warn)

|Re(z)| = |x| <= 1/em = x(max)

|Im(z)| = |y| <= ln(bn) = y(max)

If |x| is larger than x(warn), then the result will have less than half precision. If |x| is larger than x(max), then the result has no precision. Finally, if |y| is too large, the result will overflow.

cos(z) is the complex cosine of z.

|Re(z)| = |x| <= 1/sqrt(em) = x(warn)

|Re(z)| = |x| <= 1/em = x(max)

|Im(z)| = |y| <= ln(bn) = y(max)

If |x| is larger than x(warn), then the result will have less than half precision. If |x| is larger than x(max), then the result has no precision. Finally, if |y| is too large, the result will overflow.

tan(z) is the complex tangent of z. If |cos(z)|**2 is very small, that is, if x is very close to pi/2 or 3*pi/2 and if y is small, then tan(z) is nearly singular. If |cos(z)|**2 is somewhat larger but still small, then the result will be less accurate than half precision. When 2x is so large that sin(2x) cannot be evaluated to any nonzero precision, a special situation results. If |y| < 3/2, then tan cannot be evaluated

accurately to better than one significant figure. If 3/2 <= |y| < -0.5*ln(es/2), then tan can be evaluated by ignoring the real part of the argument; however, the answer will be less accurate than half precision.

arcsin(z) is the complex arc sine of z. |x| must be less than or equal 1.

arccos(z) is the complex arc cosine of z. |x| must be less than or equal to 1.

arctan(z) is the complex arc tangent of z. The argument z must not be exactly +/-i, because atan(+/-i) is undefined. In addition, z must not be so close to +/-i that substantial significance is lost.

arctan2(zn,zd) is the quadrant-correct, complex, arc tangent atan(zn/zd). The ratio z = zn/zd must not be +/-i, because atan(+/-i) is undefined. Likewise, zn and zd must not be both zero. Finally, z must not be so close to +/-i that substantial significance is lost.

sinh(z) is the hyperbolic sine of z.

|Im(z)| = |y| <= 1/sqrt(em) = y(warn)

|Im(z)| = |y| <= 1/em = y(max)

|Re(z)| = |x| <= ln(bn) = x(max)

If |y| is larger than y(warn), then the result will be less accurate than half precision. If |y| is larger than y(max), the result has no precision.

Finally, if |x| is too large, the result overflows.

cosh(z) is the hyperbolic cosine of z.

|Im(z)| = |y| <= 1/sqrt(em) = y(warn)|Im(z)| = |y| <= 1/em = y(max)

|Re(z)| = |x| <= ln(bn) = x(max)

If |y| is larger than y(warn), then the result will be less accurate than half precision. If |y| is larger than y(max), the result has no precision. Finally, if |x| is too large, the result overflows.

tanh(z) is the hyperbolic tangent of z. If |cosh(z)|**2 is very small, that is, if y mod 2*pi is very close to pi/2 or 3*pi/2 and if x is small, than tanh(z) is nearly singular. If |cosh(z)|**2 is somewhat larger but still small, then the result will be less

accurate than half precision. When 2y is so large that sin(2y) cannot be evaluated accurately to even zero precision, a special situation results. If |x| < 3/2, then tanh cannot be evaluated accurately to better than one

significant figure. If 3/2 <=|y| < - 0.5*ln(es/2), then tanh can be evaluated by ignoring the imaginary part of the agrument; however, the answer will be less accurate than half precision.

arcsinh(z) is the arc hyperbolic sine of z. Almost all arguments are legal. Only when |z| > bn/2 can an overflow occur.

arccosh(z)is the arc hyperbolic cosine of z. Almost all arguments are legal. Only when |z| > bn/2 can an overflow occur.

arctanh(z) is the arc hyperbolic tangent of z. The argument must not be exactly +/-1, because the arc hyperbolic tangent of z is undefined there. In addition, z must not be so close to +/-1 that substantial significance is lost.

1.2.9.2 Source

Proposed by Al Freed, NASA. Based on the IMSL package.

Appendix B: List of Contributors

Compiler Developers	**Email Address**
Andrew Cadach	71333.2346@compuserv.com
Paul Curtis	---
Gunter Dotzel	100023.2527@compuserv.com
John Gough	gough@fitmail.fit.qut.edu.au
Taylor Hutt	thutt@access.digex.com
Brian Kirk	robinsons@cix.compulink.co.uk
Hanspeter Mössenböck	Moessenboeck@cs.inf.ethz.ch
Alex Nedorya	ned@isi.itfs.nsk.su
Cuno Pfister	pfister@inf.ethz.ch
Josef Templ	templ@inf.ethz.ch
Rick Watson	watson@futurs.enet.dec.com

Applications Reviewers	
Steve Collins	71333.2346@compuserve.com
Al Freed	al@sarah.lerc.nasa.gov
Euan Hill	100143.1660@compuserve.com
Steve Metzeler	----
Anja Schumacher	---
Steve Terrapin	100023.1307@compuserve.com
Nick Walsh (deceased)	---

Appendix C: Oakwood Conference

Croydon 21 .. 23 June 1993

3.1 List of Contributors and Participants

Name (Initials)	Country	Organisation
Christof Brass (CB)	Switzerland	Analytic AG
Oliver Breuninger (OB)	Germany	Individual
Andrew Cadach (AC)	Russia	ISI SD RAS
Steve Collins (SC)	UK	Real Time Assoc.
Andreas Distely (AD)	Switzerland	ETH
Gunter Dotzel (GD)	Germany	ModulaWare GmbH
Dave Fox (DF)	UK	Real Time Assoc.
John Gough (JG)	Australia	QUT Carden Point
Jim Hawkins (JH)	UK	Amiga
Euan Hill (EH)	UK	BSC SIG
Stig Holmberg (SH)	Sweden	Ostersund Univ.
Wolfgang Hugentober(WH)	Switzerland	L Kissling & Co. AG
Taylor Hutt (TH)	USA	Individual
Brian Kirk (BK)	UK	Robinson Assoc.
Hans Klaver (HK)	Netherlands	Individual
Bernhard Leisch (BL)	Austria	Johannes Kepler
Steve Metzeler (SM)	Switzerland	Alchemia Software
Alex Nedorya (AN)	Russia	ISI SD RAS
Cuno Pfister (CP)	Switzerland	Oberon Microsys.
Markus Rauber (MR)	Switzerland	CATS AG
Steve Rumsby (SR)	UK	De Montford
Peter Schulthess (PS)	Germany	Ulm University
Anja Schumacher (AS)	Germany	Siemens AG
Fridtjof Siebert (FS)	Germany	Amiga Software
Josef Templ (JT)	Switzerland	ETH
Steve Terepin (ST)	UK	Opus 1 Software
Nick Walsh (NJW)	UK	City University
Rick Watson (RW)	UK	DEC

Pre-conference contributions and/or apologies received from the following who were unable to attend ...

Whitney de Vries (WV)	Canada	McGill University
J Gutknecht (JG)	Switzerland	ETH
Cheryl Lins (CL)	USA	Apple Corp.

Ian Marshall (IM)	UK	Real Time Assoc.
Michael McGaw (MM)	USA	NASA
Hanspeter Mössenböck (HM)	Switzerland	ETH
Alan Freed (AF)	USA	NASA
Chris Johnson (CJ)	USA	NASA
Niklaus Wirth (NW)	Switzerland	ETH
Dick Pountain (DP)	UK	BYTE magazine
Mark Woodman (MW)	UK	Open University

3.2 Document Modification Record

Revision	**Description**	**Date**	**Name**
0A	Initial version Emailed for comment	June 93	BK
0B	Heavily revised version based on feedback from contributors listed in Appendix B	Oct. 93	BK et al
0C	Draft reviewed by J Templ, H Mössenböck, B Kirk	Oct. 93	BK/ETH
0D	Corrections and clarifications from JT, HM and NASA Group edited in Proposed First Issue prior to ETH preface	Nov. 93	BK et al
1A	Final edits and Preface added	Dec. 93	BK/HM

3.3 Document Feedback

We would like to hear from you if you have any comments about this document or any suggestions for improving it. Please send your comments to

Oakwood Guidelines
Robinson Associates
Red Lion House
St Mary’s Street Painswick
GLOS GL6 6QR
Voice (+ 44) (0)452 813 699
Fax (+ 44) (0)452 812 912
e-Mail : robinsons@cix.compulink.co.uk

It would be very helpful if you could give specific text references where appropriate, and of course, your own name and address will make it possible to respond to your comments.

Name:

Address:

Country:Email:

Phone:Fax:

Title:

Issue:

Comments and suggestions (append additional pages if necessary) :

Date received: Date actioned: Actioned by:

DO NOT DISTRIBUTE THIS WITH THE DOCUMENT

3.4 Document Distribution Record

Revision	Organisation/name, title or location	Date & Initials
0A	To compiler developers listed in Appendix B	June 93 BK
0B	To ETH and Steve Collins	Oct. 93 BK
0C	To compiler developers listed in Appendix B	Oct. 93 BK
0D	For review by ETH prior to preface	Nov. 93 BK
1A	First public release via FTP & BCS	Dec. 93 BK