Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

A Computer System for Model Helicopter Flight Control

Technical Memo Nr. 2: The Programming Language Oberon SA

January 1998
March 1999 / Second Edition

285

A Computer System for Model Helicopter Flight Control

Technical Memo Nr. 2

The Programming Language Oberon SA

N. Wirth 21.12.97 / 15.1.99

Abstract

This memorandum provides the defining document of the programming language Oberon–SA, a subset of Oberon, extended with a few features for system– and real–time programming the ARM processor. They include array riders and interrupt handlers.

1. Introduction

It is well–known that embedded real–time applications must be based not only on effective hardware, but equally so on efficient software implementation. The engineer wishes to have close control over the program. Perhaps he may not be interested in knowing about every instruction issued by the compiler, but he must be fully aware of the code generation strategy; the compiling algorithm must be transparent. This can only be achieved by a reasonably simple language with clearly defined constructs whose representation in terms of the processor architecture is well understood.

With this in mind, we define a subset of the programming language Oberon, implemented for the processor Strong ARM (DC1035). In the following, we describe this language by stating its differences to Oberon. There are some general restrictions due to omitted features, and some minor but important extensions. The compiler is a fast single–pass system specified in Oberon.

2. Restrictions of Oberon–SA with respect to Oberon

There are a few essential restrictions and some minor ones that could also be considered as implementation restrictions. We first list the former kind.

– The most evident restriction relates to the available data types. The set of basic types has been reduced to BOOLEAN, INTEGER, REAL, and CHAR. The types SHORTINT, LONGINT, LONGREAL, and SET have been eliminated. All integers use the 32–bit format. This represents a significant simplification of the compiler as well as the task of the programmer, and it is well justified for the application on hand.

– In the area of statements, several of the constructs of lesser importance have been eliminated: LOOP, EXIT, RETURN, CASE, and WITH statements have been excluded. The symbol RETURN, used to indicate the result of a function procedure, is now syntactically coupled with the end of the procedure body (see syntax), and hence occurs exactly once in every function procedure declaration, without being considered as a statement by itself.

– Parameters called by value cannot be of a structured type (array or record). Neither can a value (expression) of an array or record structure be assigned to variables.

– Constants, variables, types, and procedures can be referenced only if they are global or strictly local.

– Variables cannot be exported and imported.

– Record types cannot be extended.

– Aliasing of imported module names is impossible.

– Strings must occur as procedure parameters only. They must be considered as constant parameters passed by reference. No assignments must be made to such parameters. This restriction avoids the need for string copying upon procedure call.

– Open array parameters must be of a single dimension only.

– Functions must not occur in expressions when intermediate results are to be saved until after return of the result.

– There exists no pseudo–module SYSTEM for standard (inline) procedures and functions. A list of available procedures is given in Section 4 below. The following standard procedures have been eliminated:

HALT CAP SIZE MIN MAX COPY SHORT LONG INCL EXCL LEN

3. Additional Facilities

We distinguish between essential additions that let the programmer express facilities which could not be handled before, and additions that "merely" serve to enhance efficiency that might otherwise have required a considerably more sophisticated compiler. Into the former category falls the feature to express interrupt handling and – to a lesser degree – the facility of riders. The latter category – to be discussed first – contains the possibility of specifying variables to be allocated in registers rather than in memory.

3.0. Compiler Option

The compiler can be called with the following parameter options:

OSA.Compile *	Compile text in marked viewer
OSA.Compile @	Compile text starting with most recent selection
OSA.Compile name0 name1 ...	Compile specified text files

File names, asterisk, and @ may be followed by a single option symbol:

"+" generate index checks for static arrays with length < 256

3.1. Leaf Procedures

A leaf procedure is a procedure that contains no procedure calls in its body. An asterisk after the symbol PROCEDURE indicates that the procedure is a leaf procedure. This allows its parameters to reside in registers. Neither need they be saved upon a further call, nor be restored upon return. As a result, however, fewer registers are available for intermediate results during the evaluation of expressions. Although the StrongArm has only 16 registers, of which 4 are dedicated to fixed purposes, this restriction has hardly been noticeable, as expressions typically have few intermediate results only. Such parameters cannot be of type REAL.

3.2. Variables in Registers

An asterisk at the end of a variable declaration indicates that the variables in the list are to be allocated in registers rather than memory. This facility is available only in leaf procedures, where the marked variables are allocated to registers like the parameters. Example:

```
PROCEDURE* P(x, y: INTEGER);
   VAR u, v: INTEGER*;
BEGIN (*x in R0, y in R1, u in R2; v in R3*)
END P;
```

The effect of allocating variables in registers is a significantly enhanced efficiency, as no storage access is involved. But as in the case of parameters in leaf procedures, this allocation decrements the number of registers available to hold intermediate results. Hence the number of register–allocated variables must be kept small. It is obvious that those variables should be chosen that are most frequently accessed.

Floating–point operations are implemented as leaf procedures using integer arithmetic in the Standard Program Library module SPL. In order to make it possible to use register variables, in particular riders (see below), in procedures that contain floating–point operations represented as calls to SPL–procedures

register–variables are also admissible in non–leaf procedures. In this case they are allocated in registers R11, R10, ... The procedure must not call on any other procedures (with the exception of SPL), because the registers used for its variables might be overwritten.

```
PROCEDURE P(x, y: INTEGER);
   VAR u, v: INTEGER*;
BEGIN (*u in R11; v in R10*)
END P;
```

In leaf procedures, registers R12 (FP) and R14 (LINK) are saved on the stack. This can be suppressed by indicating with a double asterisk that the procedure is a *fast leaf procedure.* In this case, no local variables except register variables can be declared. Fast leaf procedures were introduced for implementing floating–point operations in terms of integer arithmetic in module SPL.

3.3. Riders

In high–level programming languages, elements of arrays are selected by indexing. This involves the computation of an element's address by adding the index multiplied by the element size to the array's base address. In sequential scans, a significant gain in efficiency can be obtained by retaining the address of the last element accessed, and by obtaining the next element's address by simply adding the element size to it. In analogy to the concept of file riders in Oberon, we can imagine a rider scanning an array by moving from element to element. We therefore introduce the notion of a data type RIDER reflecting this technique, which is particularly attractive in the case of the StrongArm processor which features load and store instructions with auto increment addressing.

A rider *r* is specified as shown by the following example.

```
VAR r: RIDER T n
```

T denotes the element type of arrays to be scanned by riders of this type, and *n* is the number of elements to be advanced in each step. The default is 1. *n* must be a positive integer and is called *stride.* Note that in the case of multi–dimensional arrays (matrices), *T* denotes the "ultimate" element type, and the matrix is considered as stretched out into an array of a single dimension. *T* is restricted to basic types, and RIDER is a new keyword of the language.

Rider variables can be declared local to *leaf procedures only,* and each rider takes one register. A rider *r* is initialized, that is, associated with an array variable *a* to be scanned, by the new standard procedure *SET(r, a, i). r* then points to the element *a[i].* Access to the element referenced by the rider *r* is denoted by *r↑.* Each access advances the rider to the *n*'th next element, where *n* is specified in the rider's declaration. Note that this notation is in accordance with Oberon's pointer dereferencing, where *p↑* denotes the variable referenced by pointer *p.*

Riders can also be made to move backwards by using the denotation r↑–. (↑– stands for a down–arrow). Hence, riders are equally suitable to implement queues and stacks.

3.4. Interrupt Handlers.

A bracketed integer after the symbol PROCEDURE in a declaration indicates that the procedure is an interrupt handler. Interrupt handlers must not have any parameters, and they cannot be functions. They are not treated as leaf procedures unless requested.

```
PROCEDURE [k] IntHandler;
BEGIN ...
END IntHandler;
```

Registers R0 – R11, FP, and LINK are saved upon entry and restored upon exit. Note that the StrongArm has individual SP and LNK registers for each interrupt kind. The number k is StrongArm–specific, and it indicates the offset to be used in the return jump. It must be chosen as follows:

0 for UNDEF, SWI
4 for IRQ, FIQ, Instr Abort
8 for Data Abort
20 for FIQ (special case: no registers need be saved, R8 – R11 available, leaf)

3.5. Overriding Type Checks

In some rare situations it may be desirable to override type checking. Oberon–SA provides a facility to do so in connection with parameter passing (VAR parameters only). It is indicated by letting the formal parameter's type specification be followed by an exclamation mark. We strongly recommend to use this facility with the utmost care.

4. Predefined Procedures and Functions

4.1. Type transfer functions

Type transfer function serve to map values of one basic type into another basic type.

ODD(i)	INTEGER –> BOOLEAN
CHR(i)	INTEGER –> CHAR
ORD(ch)	CHAR –> INTEGER
FLOOR(x)	REAL –> INTEGER
FLT(i)	INTEGER –> REAL

4.2. General functions

ABS(x)	absolute value of type INTEGER or REAL
LSL(i, n)	logical shift left of integer i by n bits, 0 <= n < 32
LSR(i, n)	logical shift right
ASR(i, n)	arithmetic shift right
ROR(i, n)	rotate right (ROR 0 is special case setting carry bit in PSR)

4.3. General procedures

INC(i)	INC(i, n)	i := i+n
DEC(i)	DEC(i, n)	i := i–n
SET(r, a, i)		set rider r to denote a[i]
SKIP(r, n)		make rider r skip n elements
NEW(ptr)		record allocation on heap

4.4. Procedures for accessing memory–mapped I/O

The following procedures are used to access locations with fixed addresses, e.g. device registers:

GET(a, v)	v := Mem[a]	
PUT(a, x)	Mem[a] := x	
BIT(a, n)	test bit n at address a	
GET(offset, v, base)	v := Mem[base+offset]	*base* must be register variable
PUT(offset, x, base)	Mem[base+offset] := x	–4096 < *offset* < +4096

4.5. Procedures for accessing special registers

The following procedures are used to access processor status registers:

LDPSR(s, x)	load processor status register
	x is constant or variable, s must be 2–digit hex constant mnH
	m = 2 for CPSR, m = 6 for SPSR

	n = 1: control bits only, n = 8: flag bits only, n = 9: all bits
STPSR(s, v)	store processor status register
	v is variable, s = 0 for CPSR, s = 4 for SPSR
LDCFG(n, x)	load configuration register
	x is constant or variable, n = coprocessor register no.
STCFG(n, v)	store configuration register
	v is variable, n = coprocessor register no.
FLUSH(n)	n = 15H: flush instruction cash
	n = 16H: flush data cash
	n = 17H: flush instruction and data cash
	n = 9AH: drain write buffer

4.6. Miscellaneous functions and procedures

ADDC(z, x, y)	z := x + y + carry (register variables only)
MULD(zl, x, y)	zh,zl := x * y (Rzh = next register after Rzl)
NULL(x)	BOOLEAN; REAL x = 0 or –0
XOR(x, y)	INTEGER; bitwise exclusive or
OVFL(signed)	BOOLEAN; overflow test of preceding operation

5. The Use of Riders

We present a few examples of applications of the rider concept and show its advantage over the conventional use of indexed variables. First, consider the classical version of a procedure to sum the elements of an array:

```
PROCEDURE Sum(VAR a: ARRAY OF REAL; n: INTEGER): REAL;
   VAR s: REAL; i: INTEGER;
BEGIN s := 0.0;
   FOR i := 0 TO n–1 DO s := a[i] + s END ;
   RETURN s
END Sum;
```

The equivalent version using a rider is, assuming n > 0:

```
PROCEDURE Sum(VAR a: ARRAY OF REAL; n: INTEGER): REAL;
   VAR s: REAL; r, lim: RIDER REAL;
BEGIN s := 0.0; SET(r, a, 0); SET(lim, a, n);
   WHILE r < lim DO s := r↑ + s END ;
   RETURN s
END Sum;
```

The classical version of a procedures to compute a scalar product and its counterpart based on riders are:

```
PROCEDURE Scalar(VAR a, b: ARRAY OF REAL; n: INTEGER): REAL;
   VAR sum: REAL; i: INTEGER;
BEGIN sum := 0.0;
   FOR i := 0 TO n–1 DO sum := a[i] * b[i] + sum END ;
   RETURN sum
END Scalar;

PROCEDURE Scalar(VAR a, b: ARRAY OF REAL; n: INTEGER): REAL;
   VAR sum: REAL; ra, rb, lim: RIDER REAL;
BEGIN sum := 0.0; SET(ra, a, 0); SET(rb, b, 0); SET(lim, a, n);
   REPEAT sum := ra↑ * rb↑ + sum UNTIL ra = lim;
   RETURN sum
END Scalar;
```

In this example, the instruction count is reduced from 24 to 10, and the number of memory accesses per step from 10 to 2.

A rider may be repositioned by the further standard procedure SKIP(r, n). The rider's position is advanced by n elements without accessing the array.

When trying to apply riders to matrices, however, one discovers that this concept comes to full fruition only if we are allowed to look at matrices flattened out into linear arrays. In this connection we note that in Oberon–SA, open (dynamic) arrays are restricted to a single dimension anyway. The passing of an open array parameter in itself accomplishes this flattening. Note also that the step (stride) of a rider must be a constant specified in the rider's declaration.

6. Syntax of Oberon–SA

```
ident = letter {letter | digit}.
integer = digit {digit}.
string = "'" {character} "'".

qualident = ident ["." ident].
selector = {"." ident | "[" expression {"," expression} "]" | "↑"}.
factor = qualident selector | integer | string | "TRUE" | "FALSE" | "NIL" |
          "(" expression ")" | "~" factor.
term = factor {("*" | "DIV" | "MOD" | "&") factor}.
SimpleExpression = ["+"|"–"] term {("+"|"–" | "OR") term}.
expression = SimpleExpression [("=" | "#" | "<" | "<=" | ">" | ">=") SimpleExpression].
assignment = qualident selector ":=" expression.
ActualParameters = "(" [expression {"," expression}] ")" .
ProcedureCall = qualident [ActualParameters].
IfStatement = "IF" expression "THEN" StatementSequence
          {"ELSIF" expression "THEN" StatementSequence}
          ["ELSE" StatementSequence] "END".
WhileStatement = "WHILE" expression "DO" StatementSequence "END".
RepeatStatement = "REPEAT" StatementSequence "UNTIL" expression.
ForStatement = "FOR" ident ":=" expression "TO" expression ["BY" expression] "DO"
          StatementSequence "END".
statement = [assignment | ProcedureCall | IfStatement |
          WhileStatement | RepeatStatement | ForStatement].
StatementSequence = statement {";" statement}.

ArrayType = "ARRAY" expression {"," expression} "OF" type.
FieldList = [identdef {"," identdef} ":" type].
RecordType = "RECORD" FieldList {";" FieldList} "END".
PointerType = "POINTER" "TO" (qualident | RecordType).
type = qualident | ArrayType | RecordType | PointerType.
identdef = ident ["*"].
FormalType = ["ARRAY" "OF"] qualident.
FPSection = ["VAR"] ident {"," ident} ":" FormalType.
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
ProcedureHeading = "PROCEDURE" identdef [FormalParameters].
ProcedureBody = declarations ["BEGIN" StatementSequence ["RETURN" expression]] "END".
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
declarations = ["CONST" {identdef "=" expression ";"}]
          ["TYPE" {identdef "=" type ";"}]
          ["VAR" {identdef {"," identdef} ":" type ";"}]
          {ProcedureDeclaration ";"}.
import = "IMPORT" ident {"," ident} ";".
module = "MODULE" ident ";" [import] declarations
          ["BEGIN" StatementSequence] "END" ident "." .
```