

CFB Software

Astrobe

Oberon Programmers Guide

Astrobe for RISC5

This document is applicable to Astrobe for RISC5. It shows Oberon programmers how the Astrobe implementation of Oberon differs from the standard Programming Language Oberon report. It also clarifies the details of some features which are intentionally left undefined by the report. Guidelines and examples of recommended Oberon coding techniques are included.

Astrobe Oberon Programmers Guide

Table of Contents

1	Introduction	4
2	Data Types.....	4
2.1	BYTE	4
2.2	CHAR	5
2.3	INTEGER	5
2.4	REAL	5
2.5	SET.....	5
3	Extensions	6
3.1	Array of BYTE.....	6
3.2	FOR loops	6
3.3	Built-in Procedures.....	7
3.3.1	BITS	7
3.3.2	BFI	7
3.3.3	BFX.....	8
3.4	Leaf Procedures	8
3.5	SYSTEM Procedures	9
3.5.1	PUT and GET with auto-increment / decrement.....	9
3.6	Interrupt Handlers	9
4	Restrictions.....	10
4.1	Constant declarations	10
4.2	Procedure Variables.....	10
4.3	Anonymous Pointer Types	10
4.3.1	Extensions.....	10
4.3.2	Self References	10
4.4	Direct and Indirect Imports.....	10
4.5	Export.....	10
4.6	PACK(x, e).....	11
4.7	UNPK(x, e).....	11
5	Implementation Size Limits.....	12
6	CASE Statements	13
6.1	Numeric CASE Statements	13
6.2	Numeric CASE Error Reporting.....	14
6.3	Type Extension CASE Statements	15
7	Programming Conventions and Guidelines.....	15

7.1	Essentials.....	15
7.1.1	Precondition Checks	16
7.1.2	Global Variables.....	16
7.1.3	Function Procedures.....	16
7.2	Indentation	16
7.3	Semicolons	17
7.4	Dereferencing	18
7.5	Letter case.....	18
7.6	Names	18
7.7	White space	19
7.8	Alignment.....	19
7.9	Boolean Expressions	20
7.10	Acknowledgements.....	20

1 Introduction

The RISC5 compiler included in Astrobe implements the Oberon language as defined in the included report titled:

The Programming Language Oberon (Revision 1.10.2013 / 3.5.2016) by Niklaus Wirth.

This document applies to Astrobe for RISC5. It shows Oberon programmers how the Astrobe implementation of Oberon differs from the standard Programming Language Oberon report. It also clarifies the implementation-specific details of some features which are intentionally left undefined by the report. Guidelines and examples of recommended Oberon coding techniques are included.

2 Data Types

2.1 BYTE

The BYTE type is primarily intended to be used when transferring 8- and 16-bit data to and from peripheral devices. Although BYTE variables can be used wherever an INTEGER variable is allowed (except where noted here) INTEGERS should always be used unless there is a compelling reason to do otherwise.

BYTE is an unsigned integer with a minimum value of 0 and a maximum value of 255.

BYTE variables are compatible with INTEGER variables in assignments, parameter passing and as return values from procedures. No overflow checking is performed at runtime. The following code could be used to trap runtime errors when assigning an integer value to a BYTE variable:

```
PROCEDURE IntToByte(intVal: INTEGER): BYTE;
BEGIN
  ASSERT(LSR(intVal, 8) = 0);
  RETURN intVal;
END IntToByte;
```

When using SYSTEM.PUT to store a value at a particular absolute memory location the type of the variable passed to the *SYSTEM.PUT* function determines whether a *store register byte* or *store register word* instruction is used to perform the transfer. Normally, if you specify a numeric constant value, SYSTEM.PUT will use a word-sized transfer as it interprets the constant as an INTEGER. If you want it to use a byte-sized transfer instead you should use a character constant or BYTE variable, whichever you prefer:

```
VAR
  addr: INTEGER;
  b: BYTE;
  ...
BEGIN
  SYSTEM.PUT(addr, 0X);
  SYSTEM.PUT(addr, CHR(0));
  b := 0;
  SYSTEM.PUT(addr, b);
  ...
```

2.2 CHAR

The characters of the Latin-1 set.

2.3 INTEGER

The range of valid INTEGER numbers is:

$$\text{INTEGER} = -2147483647 \dots +2147483647$$

LONGINT is an alias for INTEGER.

2.4 REAL

The range of valid REAL numbers is:

$$\text{REAL} = -3.40282\text{E}+38 \dots +3.40282\text{E}+38$$

LONGREAL is an alias for REAL.

2.5 SET

The sets of integers between 0 and 31.

3 Extensions

The primary motive for introducing extensions to the Astrobe implementation of RISC5 Oberon is to simplify the task of porting Astrobe applications from other processors i.e. ARM Cortex-M3, M4 and M7 which already have these features.

3.1 Array of BYTE

If the formal parameter is an array of bytes with a fixed size it can accept actual parameters of any type whose size is the same number of bytes e.g.

```
TYPE
  Buffer = ARRAY 256 OF BYTE;
  IntArray = ARRAY 64 OF INTEGER;
  Data = ARRAY 12 OF INTEGER;

VAR
  ia: IntArray;
  d: Data;

PROCEDURE SendData(bytes: Buffer);
...
...
SendData(ia); (* OK *)
SendData(d); (*Error: incompatible parameters *)
```

3.2 FOR loops

The control variable in a FOR loop is a read-only variable in the body of the FOR loop. For example:

```
FOR i := 0 TO 10 DO
  i := i - 1 (* Error: read-only *)
END;
```

Note that the limit of a FOR loop is evaluated on each iteration of the loop. It is the programmer's responsibility to ensure that the limit is not modified during the execution of the loop.

If the limit is a non-trivial function then it should be assigned to a local variable first. For example:

```
strlen := Strings.Length(s);
FOR i := 0 TO strlen - 1 DO
  s[i] := CAP(s[i])
END;
```

3.3 Built-in Procedures

3.3.1 BITS

```
PROCEDURE BITS(i: INTEGER): SET;
```

BITS takes an INTEGER parameter and returns a SET with the same bit pattern. It is a type transfer function like ORD rather than a type conversion function.

By definition, the following expressions, where *i* is an INTEGER and *s* is a SET, are TRUE:

```
BITS(i) = SYSTEM.VAL(SET, i)
ORD(BITS(i)) = i
BITS(ORD(s)) = s
```

BITS is convenient to use in expressions which are a mixture of INTEGERS, masks and bit fields.

Note that SYSTEM.VAL can still be used if you want compatibility with other Oberon systems.

The following examples show the use of BITS with a constant value and the equivalent SET constants:

```
BITS(0) = {}
BITS(1) = {0}
BITS(3) = {0, 1}
BITS(0FFFFFFFFH) = {0..31}
```

3.3.2 BFI

```
PROCEDURE BFI*(VAR word: INTEGER; msb, lsb, bitfield: INTEGER);
```

```
PROCEDURE BFI*(VAR word: INTEGER; bitNo, bitfield: INTEGER);
```

BFI updates a bitfield, i.e. just a portion of 32-bit word, with an INTEGER value. *word* is the target variable and *bitfield* is the source data. *msb* and *lsb* are constant values. *msb* is the most-significant bit and *lsb* is the least-significant bit of the bitfield.

If *msb* = *lsb* (i.e. only a single-bit is accessed) then the two parameters can be replaced by the single *bitNo*.

Examples of its use can be seen in the realtime clock (RTC) library module. Clock data are packed into a single word as follows:

No of Bits:	6	4	5	5	6	6
Bit Field:	31..26	25..22	21..17	16..12	11..6	5..0
Data:	Year	Month	Day	Hour	Minute	Second

e.g. the following statements update just the *hours* and *minutes* values stored in the datetime variable *dt*; the *seconds*, *day*, *month* and *year* values remain unchanged:

```
BFI(dt, 11, 6, mins);
BFI(dt, 16, 12, hrs);
```

3.3.3 BFX

```
PROCEDURE BFX(VAR word: INTEGER; msb, lsb): INTEGER;
```

```
PROCEDURE BFX(VAR word: INTEGER; bitNo: INTEGER): INTEGER;
```

BFX returns an unsigned *bitfield*, i.e. just a portion of 32-bit word, from an INTEGER value. *word* is the source data. *msb* and *lsb* are constant values. *msb* is the most-significant bit and *lsb* is the least-significant bit of the bitfield portion of the word.

If *msb* = *lsb* (i.e. only a single-bit is accessed) then the two parameters can be replaced by the single *bitNo*.

Examples of its use can be seen in the realtime clock (RTC) library module. Clock data are packed into a single word as follows:

No of Bits:	6	4	5	5	6	6
Bit Field:	31..26	25..22	21..17	16..12	11..6	5..0
Data:	Year	Month	Day	Hour	Minute	Second

e.g. the following statements extract just the month and year values from the datetime variable *dt*:

```
mm := BFX(dt, 25, 22);  
yy := BFX(dt, 31, 26);
```

3.4 Leaf Procedures

The code that is generated by the Oberon compiler for procedure calls is efficient for most normal purposes. On occasions where faster execution speed is required (e.g. for fast interrupts) *Leaf* procedures can be used. These are identified by an asterisk in the procedure header.

```
PROCEDURE* GetValue(VAR n: INTEGER);
```

Leaf procedures result in faster execution speed because the following runtime checks are disabled:

- Array index bounds
- Invalid type guards
- Bad INTEGER divisors
- NIL pointers

Limitations of leaf procedures are:

- Procedures (other than standard and SYSTEM procedures) cannot be called from a leaf procedure
- Some runtime errors go undetected

Although the standard procedures *ODD*, *CHR* etc. and SYSTEM procedures *PUT*, *GET* etc. look like normal procedures they are implemented as inline code so they can be used in leaf procedures.

The following examples illustrate the difference between an asterisk used to indicate that a procedure is a leaf procedure and an asterisk used to indicate that the procedure is exported:

```
PROCEDURE GetValue(VAR n: INTEGER);    (* Private non-leaf procedure *)
PROCEDURE GetValue*(VAR n: INTEGER);  (* Exported non-leaf procedure *)

PROCEDURE* GetValue(VAR n: INTEGER);  (* Private leaf procedure *)
PROCEDURE* GetValue*(VAR n: INTEGER); (* Exported leaf procedure *)
```

3.5 SYSTEM Procedures

3.5.1 PUT and GET with auto-increment / decrement

An optional integer constant parameter *inc* can be used with the functions SYSTEM.GET and SYSTEM.PUT to automatically increment / decrement the value of the address of the variable that is being accessed.

```
PROCEDURE GET*(address: INTEGER; VAR v: <any basic type>);
PROCEDURE GET*(VAR address: INTEGER; VAR v: <any basic type>; inc: INTEGER);
PROCEDURE PUT*(address: INTEGER; x: <any basic type>);
PROCEDURE PUT*(VAR address: INTEGER; x: <any basic type>; inc: INTEGER);
```

If *inc* is in the range 1..255 the address is incremented after the value is accessed. If *inc* is in the range -1..-255 the address is decremented before the value is accessed. Typical values for *inc* are 1 for byte accesses and 4 for word accesses.

3.6 Interrupt Handlers

An Oberon interrupt handler is a normal procedure which has an integer constant in square brackets instead of a list of parameters. The constant can be a literal or named constant e.g.

```
PROCEDURE TimerHandler[0];

or

CONST
  IRQ = 0;

PROCEDURE Timer1Handler[IRQ];
```

The value of the constant is currently unused. Its presence is required to enable the compiler to distinguish interrupt handler procedures from normal parameterless procedures.

Refer to the example module *TestInt* to see how an interrupt handler can be used.

4 Restrictions

4.1 Constant declarations

REAL expressions are not allowed in CONST declarations.

4.2 Procedure Variables

It is an error if a procedure variable is assigned a procedure which is declared in a different scope. However, the compiler does not report the error.

4.3 Anonymous Pointer Types

4.3.1 Extensions

Pointer types declared in an external module can only be extended in a client module if they point to a named record.

e.g. the type *Item* declared as:

```
Item* = POINTER TO RECORD value: INTEGER END;
```

can be extended in a client module if the declaration is changed to the equivalent form:

```
Item* = POINTER TO ItemDesc;  
ItemDesc* = RECORD value: INTEGER END;
```

4.3.2 Self References

Anonymous pointer types cannot directly reference themselves.

e.g. the type *Item* declared as:

```
Item = POINTER TO RECORD value: INTEGER; next: Item END;
```

should be changed to the equivalent form:

```
Item = POINTER TO ItemDesc;  
ItemDesc = RECORD value: INTEGER; next: Item END;
```

4.4 Direct and Indirect Imports

A client module requires an imported module to precede any module which indirectly imports it, in the IMPORT list. For example, if B imports A, and C imports both A and B, then A must appear before B in C's IMPORT list. The error is detected and reported with a compiler error message.

4.5 Export

- String constants and anonymous variables cannot be exported.
- Pointer types can only be exported if the base type is also exported.
- Pointer variables can only be exported if the pointer type is also exported.
- Parameters of exported procedures must be of exported data types.

4.6 PACK(x, e)

Prerequisite: $1.0 \leq x < 2.0$

4.7 UNPK(x, e)

Prerequisite: $x \geq 0.0$

5 Implementation Size Limits

Items	Maximum
Type extension levels	3
Parameters to a procedure	11
Minimum value of a CASE label	0
Maximum value of a CASE label	255
Number of labels in a CASE statement	256
Code size of a module	32000 bytes
Number of characters in a string constant	256 (including the terminating null)

6 CASE Statements

6.1 Numeric CASE Statements

In the RISC5 Oberon compiler, the numeric CASE statement has been implemented in a way that provides maximum speed and predictability of code-generation at the expense of memory consumption.

For best results, restrict the use of numeric CASE statements to situations where:

- The case labels are naturally bytes, integers or characters
- The case labels are relatively contiguous
- There are a large number of cases (i.e. more than half-a-dozen)
- All cases have similar probabilities of occurrence

Otherwise consider using an IF-ELSIF...ELSIF-ELSE series of statements instead.

In some cases a hybrid combination of CASE and IF statements can result in a good compromise between readability, efficiency and memory usage.

Consider the following example which could be used to map a set of strings to a corresponding integer code:

```
PROCEDURE FindKeyword*(id: ARRAY OF CHAR; VAR sym: INTEGER);
BEGIN
  sym := ident;
  IF id = "ARRAY" THEN sym := array
  ELSIF id = "BEGIN" THEN sym := begin
  ELSIF id = "BY" THEN sym := by
  ELSIF id = "CASE" THEN sym := case
  ELSIF id = "CONST" THEN sym := const
  ELSIF id = "DIV" THEN sym := div
  ...
  ...
```

You can write this more efficiently with a hybrid combination of CASE and IF-THEN as follows:

```
PROCEDURE FindKeyword*(id: ARRAY OF CHAR; VAR sym: INTEGER);
BEGIN
  sym := ident;
  CASE id[0] OF
    "A":
      IF id = "ARRAY" THEN sym := array
      END |
    "B":
      IF id = "BEGIN" THEN sym := begin
      ELSIF id = "BY" THEN sym := by
      END |
    "C":
      IF id = "CASE" THEN sym := case
      ELSIF id = "CONST" THEN sym := const
      END |
    "D":
      IF id = "DIV" THEN sym := div
      ELSIF id = "DO" THEN sym := do
      END |
  ...
  ...
```

Timing tests using an example with ~30 cases, assuming each word occurs with the same frequency, indicates that the CASE solution is 4 times faster than the IF-THEN ladder solution. However, the CASE approach generates 5% more code.

If you do use the IF-THEN ladder you should check for matches that you expect to occur most frequently in the first comparisons and those that you expect to occur least frequently in the last comparisons for best efficiency.

6.2 Numeric CASE Error Reporting

The following CASE statement errors are trapped and reported:

- Duplicate CASE labels are reported as compile-time errors.
- A reference to a missing label results in a runtime error and program termination.
- The type of the selector must be BYTE, INTEGER or CHAR
- The type of each label must be type-compatible with the selector.

You should design your programs so that any conditions not satisfied by the CASE statement are handled separately, as illustrated in the following example:

```
PROCEDURE ToUpperCase(VAR ch: CHAR);
BEGIN
  IF (ch >= "a") & (ch <= "z") THEN
    ch := CHR(ORD(ch) - ORD("a") + ORD("A"))
  END
END ToUpperCase;

PROCEDURE SoundexCode(ch: CHAR): INTEGER;
VAR
  value: INTEGER;
BEGIN
  ToUpperCase(ch);
  IF (ch < "A") OR (ch > "Z") THEN
    value := 0
  ELSE
    CASE ch OF
      "A", "E", "I", "O", "U", "H", "W", "Y":
        value := 0 |
      "B", "F", "P", "V":
        value := 1 |
      "C", "G", "J", "K", "Q", "S", "X", "Z":
        value := 2 |
      "D", "T":
        value := 3 |
      "L":
        value := 4 |
      "M", "N":
        value := 5 |
      "R":
        value := 6
    END
  END;
  RETURN value
END SoundexCode;
```

6.3 Type Extension CASE Statements

Note that the syntax definition for the type test form of the CASE statement is:

```
CaseStatement = CASE qualident OF case {"|" case} END.  
case = [qualident ":" StatementSequence]
```

This differs from the numeric form of the CASE statement:

```
CaseStatement = CASE expression OF case {"|" case} END.  
case = [CaseLabelList ":" StatementSequence].  
CaseLabelList = LabelRange {"|" LabelRange}.  
LabelRange = label [".." label].  
label = integer | string | qualident.  
WhileStatement = WHILE expression DO
```

and the IS form of type test:

```
expression = SimpleExpression [relation SimpleExpression].  
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
```

both of which allow the type of an *expression* rather than a *qualified identifier* to be tested. Consequently, valid examples of type tests, using the type definitions in `ExtensionsCase` example supplied with `Astrobe`, are:

```
Shape: Shape;  
shapes: ARRAY 4 OF Shape;  
  
shape := shapes[1];  
CASE shape OF  
  Rectangle: shape.width := w;  
  ...  
  ...  
  
IF shape[1] IS Rectangle THEN  
  shape[1](Rectangle).width := w;
```

and an invalid example of a type test is:

```
CASE shape[1] OF  
  Rectangle: shape[1].width := w;  
  ...  
  ...
```

7 Programming Conventions and Guidelines

This chapter describes the programming guidelines and source code formatting conventions which have been used in software developed using `Astrobe`.

Some programming guidelines are more important than others. In the first section, the more important ones are described. The remaining sections contain more cosmetic rules which describe the look-and-feel of Oberon programs published by CFB Software. If you like them, feel free to use them for your programs as well. It may make your programs easier to understand for someone who is used to the design, documentation, and coding patterns used in applications developed using `Astrobe`.

7.1 Essentials

The most important programming conventions all centre around the aspect of evolvability. It should be made as easy as possible to change existing programs in a reliable way, even if the program has been written a long time ago or by someone else. Evolvability can often be

improved by increasing the locality of program pieces: if a piece of program may only have an effect on a clearly locatable stretch of program text, it is easier to know where a program modification may necessitate further changes. Basically, it's all a matter of keeping "ripple effects" under control.

7.1.1 Precondition Checks

Preconditions are one of the most useful tools to detect unaccounted ripple effects. Precondition checks allow to pinpoint semantic errors as early as possible, i.e. as closely to their true source as possible. After larger design changes, properly used assertions can help to dramatically reduce debugging time.

Whenever possible, use static means to express what you know about a program's design. In particular, use the type and module systems of Oberon for this purpose; so the compiler can help you to find inconsistencies, and thus can become an effective refactoring tool.

Precondition assertions should be used consistently. Don't allow client code to "enter" your module if it doesn't fulfil the preconditions of your module's procedures. In this way, you avoid propagation of foreign errors into your own code.

```
PROCEDURE Ten*(e: INTEGER): REAL;  
BEGIN  
  ASSERT((e >= 0) & (e <= 38), 21)  
  ...  
END
```

Assertion codes should be in the range 100 to 255 to avoid being confused with those used in the Astrobe runtime system and libraries.

7.1.2 Global Variables

There should be as few global variables as possible. Global variables can be accessed from many places in a program, at different times. This makes it difficult to keep track of all possible interactions ("side effects") with such variables. This in turn increases the likelihood of introducing errors when changing the use of them.

7.1.3 Function Procedures

Procedures which return a result should not modify global variables or VAR parameters as side effects. It is easier to deal with function procedures if they are true functions in the mathematical sense, i.e., if they don't have side effects. Returning function results is ok.

Procedures should be kept as small as is practicable. It is preferable if the whole function is visible on the screen without having to scroll.

7.2 Indentation

A new indentation level is realised by pressing the tab key. The number of spaces inserted depends on the editor option *Indent width*.

A monotype font (e.g. Times New Roman, Consolas) should be used to assist consistent indentation.

Do not use more than three levels of nesting (IF, WHILE etc.). Aim to limit the scope of each block statement so that it is completely visible on one screen.

Combine nested IFs into single boolean expressions where appropriate:

```
IF (p # NIL) THEN
  IF (p.val # 0) THEN
```

should be written as:

```
IF (p # NIL) & (p.val # 0) THEN
```

Oberon uses short-circuit evaluation of such expressions, i.e. if the first expression is FALSE, the second expression is not evaluated.

7.3 Semicolons

Semicolons are used to separate statements, not to terminate statements. This means that there should be no superfluous semicolons.

Good

```
IF done THEN
  Print(result)
END
```

Bad

```
IF done THEN
  Print(result);
END
```

7.4 Dereferencing

The optional dereferencing operator `^` should be left out wherever possible.

Good

```
h.next := p.prev.next
```

Bad

```
h^.next := p^.prev^.next
```

7.5 Letter case

In general, each identifier starts with a small letter, except:

- A module name always starts with a capital letter
- A type name always starts with a capital letter
- A procedure always starts with a capital letter, this is true for procedure constants, types, variables, parameters, and record fields.

Good

```
null = 0X;  
DrawDot = PROCEDURE (x, y: INTEGER);  
PROCEDURE Proc (i, j: INTEGER; Draw: DrawDot);
```

Bad

```
NULL = 0X;  
PROCEDURE isEmpty (q: Queue): BOOLEAN;  
R = RECORD  
  draw: DrawDot  
END;
```

Don't capitalise identifiers with more than one character. They should be reserved for the language. An exception is when you use peripheral register names in your programs that are consistent with those used in the MCU manufacturers' documentation e.g. *MCU.PINSEL1*

7.6 Names

- A proper procedure has a verb as name, e.g. *DrawDot*
- A function procedure has a noun or a predicate as name, e.g. *Exponent(r)*, *IsEmpty(q)*
- Procedure names which start with the prefix *Init* are snappy, i.e., they have an effect only when called for the first time. If called a second time, a snappy procedure either does nothing, or it halts. In contrast, a procedure which sets some state and may be called several times starts with the prefix *Set*.
- *CamelCaps* should be used to identify each word in an identifier, e.g. *startAddress* not *startaddress*
- Names should not be unnecessarily long nor unnecessarily abbreviated, e.g. *maxStep* not *maximumForLoopStep*, *nextPage* not *nxtpg* etc.

7.7 White space

A single space should be inserted between lists of symbols, between actual parameters, and between operators:

Good

```
VAR a, b, c: INTEGER;
DrawRect(1, t, r, b);
a := i * 8 + j - m[i, j];
```

Bad

```
VAR a,b,c: INTEGER;
DrawRect(1,t,r,b);
a:=b;
a := i*8 + j - m[i,j];
```

7.8 Alignment

- Opening and closing keywords are either aligned or on the same line
- IMPORT, CONST, TYPE, VAR, PROCEDURE sections are one level further indented than the outer level.
- PROCEDURE X and END X are always aligned
- If the whole construct does not fit on one line, there is never a statement or a type declaration after a keyword
- The contents of IF, WHILE, REPEAT, FOR, CASE constructs are one level further indented if they do not fit on one line.

Good

```
IF expr THEN S0 ELSE S1 END;
REPEAT S0 UNTIL expr;
WHILE expr DO S0 END;

IF expr THEN
  S0
ELSE
  S1
END;

REPEAT
  S0
UNTIL expr;

i := 0; WHILE i # 15 DO DrawDot(a, i); INC(i) END;

TYPE Square = POINTER TO RECORD(Rectangle) END;

IMPORT Lists, Out,
  Reals, Main;

VAR
  proc: Lists.Proc;
```

Bad

```
IF expr THEN S0
ELSE S1 END;

PROCEDURE P;
BEGIN ... END P;

BEGIN i := 0;
  j := a + 2;
  ...

REPEAT i := 0;
  j := a + 2;
```

7.9 Boolean Expressions

Boolean expressions are often misused. Complex logical expressions can often be reduced to a simpler form. Use truth tables to confirm that the simpler form is equivalent.

```
IF (~summary) OR (summary & ~printing)
```

can be simplified to:

```
IF ~(summary & printing)
```

Some transformations reveal that two booleans are essentially equivalent and one can be removed altogether.

```
IF continue THEN finished := FALSE ELSE finished := TRUE END;
```

should just be:

```
finished := ~continue;
```

NOTE: DO NOT be tempted to make the same transformation to the statement:

```
IF continue THEN finished := FALSE END;
```

Finally,

```
IF continue = TRUE THEN
```

should just be:

```
IF continue THEN
```

7.10 Acknowledgements

The guidelines in this chapter have been adapted from the original *BlackBox Component Builder Programming Conventions* with the kind permission of Oberon microsystems AG. (www.oberon.ch)