CFB Software

Astrobe

ARM Cortex-M Oberon Programmers Guide

This document is generally applicable to all ARM Cortex-M editions of Astrobe. 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

The ARM Cortex-M 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 is generally applicable to all editions of Astrobe for Cortex-M. 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 Vocabulary

2.1 Underscore Characters in Identifier Names

Underscore characters (_) are allowed in identifier names (i.e. constant definitions etc.). This feature is solely intended for use with *multi-word uppercase* identifiers. Normally uppercase names should only be used for Oberon's reserved words and standard procedures and *CamelCaps* should be used to distinguish separate words in your own identifier names. However a justifiable exception to this rule is the use of uppercase peripheral register names in your programs to match those used by microcontroller manufacturers in their documentation. For example:

```
MCU.PINMODEOD0
```

can be defined as:

```
MCU.PINMODE_OD0
```

3 Constants and Types

3.1 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 on BYTE variables 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;
```

Constants in the range 0..255 can be assigned to BYTE variables, passed to BYTE parameters and returned as BYTE values from functions. Attempts to use a constant outside this range where a BYTE value is expected will result in the compile-time error: *out of range*.

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* (STRB) or *store register word* (STR) 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);
  ...
```

3.2 CHAR

The characters of the Latin-1 set.

3.3 INTEGER

The range of valid INTEGERs allowed is -2^{31} to $+2^{31}-1$. In the following description these values are referred to as MinInt (-2147483648) and MaxInt (+2147483647).

A number of MinInt-related anomalies exist in the current implementation:

- The compiler does not report an error if 2147483648 (MaxInt+1) is specified as a constant. The actual value stored is MinInt.
- ABS(MinInt) = MinInt
- -MinInt = MinInt

LONGINT is an alias for INTEGER.

3.4 REAL

The range of valid REAL numbers is:

```
REAL      -3.40282E+38 .. +3.40282E+38
```

LONGREAL is an alias for REAL.

3.5 SET

The sets of integers between 0 and 31.

4 Extensions

4.1 Array of BYTE

If a formal parameter to a procedure is defined as ARRAY OF BYTE the actual parameter may be of any data type. The parameter can then be accessed byte-by-byte in the body of the procedure e.g.

```
VAR
  data: INTEGER;
  b1, b2, b3, b4: BYTE;

PROCEDURE WordToBytes(w: ARRAY OF BYTE; VAR b1, b2, b3, b4: BYTE);
BEGIN
  ASSERT(LEN(w) = 4, 20);
  b1 := w[0];
  b2 := w[1];
  ...
  ...

WordToBytes(data, b1, b2, b3, b4);
```

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 *)
```

4.2 Local Dynamic Arrays

A local dynamic array is an array of any type declared within a procedure. The number of elements (i.e. length) of the array is determined at runtime. The syntax is:

```
ArrayType = "ARRAY OF" type.
```

For example:

```
TYPE
  MonthlyEvent = RECORD
    date: Date;
    time: Time;
    location: Location
  END;
  YearEvents = ARRAY 12 OF MonthlyEvent;

PROCEDURE P();
VAR
  counts: ARRAY OF INTEGER;
  weights: ARRAY OF REAL;
  months: ARRAY OF MonthlyEvent;
  years: ARRAY OF YearEvents;
  ...
  ...
```

The length of a local dynamic array is established by a call to NEW with a second parameter indicating the desired number of array elements. For example:

```
NEW(counts, len)
```

where *len* is a non-negative INTEGER expression.

The great advantage of local dynamic arrays is that they can be safely allocated on the stack. Thus no garbage collector or special treatment is required to free their memory when the procedure terminates. This allows the most efficient use of memory without the associated problems of memory leaks and fragmentation.

For example, given the following procedure, the call *DisplayHighest(20, 200)* would read the next 200 integer data items and display the top 20 values read. *DisplayHighest(20, 20)* would just display the next 20 values in ascending order.

```
PROCEDURE DisplayHighest(selection, total: INTEGER);
VAR
  data: ARRAY OF INTEGER;
BEGIN
  ASSERT(selection <= total, 100);
  NEW(data, total);
  FOR i := 0 TO total - 1 DO ReadInteger(data[i]) END;
  Sort.Integers(data);
  FOR i := 0 TO selection - 1 DO Out.Int(data[i], 6); Out.Ln END
END DisplayHighest;
```

When the procedure terminates, the 800 bytes (or 80 bytes in the second case) of RAM allocated to the data array is automatically returned to the system to be reused.

NOTE: This implementation of dynamically allocated arrays is subject to the restrictions that they are one-dimensional and cannot be elements of other data structures.

4.3 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;
```

4.4 Built-in Procedures

4.4.1 ABS

```
PROCEDURE ABS(s: SET): INTEGER;
```

An overloaded form of the standard procedure ABS takes a SET parameter and returns the number of elements in the set. For example:

```
ABS({}) = 0
ABS({0}) = 1

s := {1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31};
ABS(s) = 11

s := {0..15};
ABS(s + {16..31}) = 32
```

4.4.2 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. For example, in the Astrobe library procedure *Serial.SetFormat*, to modify the wordlength in the ULCR register we have to clear the six least significant bits of ULCR and replace them with the integer value: (wordlength - 5). In earlier versions of Astrobe, one way to do this would be to write:

```
PROCEDURE SetFormat*(wordLength, stopBits: INTEGER; parity: SET);
VAR
  ULCR: SET;
BEGIN
  SYSTEM.GET(MCU.U0LCR, ULCR);
  ULCR := ULCR - {0..5} + parity + SYSTEM.VAL(SET, wordLength - 5));
```

Using BITS, this last statement simplifies to:

```
ULCR := ULCR - {0..5} + parity + BITS(wordLength - 5);
```

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}
```

4.4.3 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 (Clock) library module e.g. the following statement updates just the *minutes* values stored in *time*; the *hours* and *seconds* values remain unchanged:

```
BFI(time, 11,  8, mm MOD 10);    (* time:11:8 := minutes units *)

BFI(time, 14, 12, mm DIV 10);    (* time:14:12 := minutes tens *)
```

4.4.4 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 (Clock) library module e.g. the following statement extracts just the tens and units values of minutes from time:

```
 mm  := 10 * BFX(time, 14, 12) + BFX(time, 11,  8);
```

4.4.5 DISPOSE

```
PROCEDURE DISPOSE(VAR ptr: <Pointer Type>);
```

DISPOSE is a built-in function which frees memory which has previously been allocated to a pointer record / array with NEW. The default behaviour of DISPOSE can be changed by substituting a user-defined function for the default library function *Storage.Deallocate*.

4.4.6 LSR

```
PROCEDURE LSR(x, n: INTEGER): INTEGER;
```

LSR performs an unsigned shift right of integer x by integer expression n places, returning x * 2^{-n}

4.5 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* Speedy(n: INTEGER);
```

Features of leaf procedures that result in faster execution speed are:

- Parameters are stored in registers
- Registers do not need to be saved or restored
- INTEGER and SET local variables are stored in registers
- Array index checks are suppressed
- Procedure overhead is as little as two instructions if no stack space is used

Limitations of leaf procedures are:

- Procedures (other than standard and SYSTEM procedures) cannot be called from a leaf procedure
- REAL operations are restricted for systems that do not have a hardware FPU (e.g. Cortex-M3)
- Array index out of range errors are not detected
- As there are a limited number of microcontroller registers available, there is a limit to the combined total of parameters and local variables and the complexity of the code that can be used in a leaf procedure.

Although the standard procedures *ODD, CHR* etc. and SYSTEM procedures *PUT, GET* etc. look like normal procedures most are implemented as inline code so they can be used in leaf procedures. The exceptions are the standard procedures *FLT, FLOOR* and *NEW* (when used to allocate heap memory). These are implemented as procedure calls so cannot 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 *)
```

4.6 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**.**

An example of an IRQ procedure used to handle interrupts from the LPC1343 timer *Timer1* is:

```
   PROCEDURE TimerHandler[0];
   BEGIN
     INC(mSec);
     (* Reset match register 0 interrupt*)
     SYSTEM.PUT(MCU.TMR32B1IR, {0})
   END TimerHandler;
```

The handler increments the global variable *mSec.* It then resets the timer ready to handle the next interrupt.

An example of the code required to install this handler using the *Assign* procedure from the *Traps* module is:

```
  CONST
    Timer1ExceptionVector = 010000048H;

    Traps.Assign(Timer1ExceptionVector, TimerHandler);
```

The source code of *IRQBlinker.mod* and *IRQTimer.mod* is included in the Astrobe examples folders. These form a complete working example of a timer interrupt-driven blinking LED.

5 Restrictions

5.1 Constant declarations

REAL expressions are not allowed in CONST declarations.

5.2 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.

5.3 Anonymous Pointer Types

5.3.1 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;
```

5.3.2 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;
```

5.4 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.

5.5 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.

5.6 PACK(x, e)

Prerequisite: 1.0 <= x < 2.0

5.7 UNPK(x, e)

Prerequisite: x >= 0.0

6 Implementation Size Limits

Items	Maximum
Type extension levels	4
Types exported from a module	17
Number of modules in an IMPORT list	32
Number of modules in an application	256
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 CASE statement	32KB
Step size in a FOR statement	-256 <= step < 255
Number of characters in a string constant	256 (including the terminating null)
Global variable allocation in a module	32KB

7 CASE Statements

7.1 Numeric CASE Statements

In the ARM 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. The maximum size of the code that can be contained in a single CASE statement is 32KB.

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.

7.2 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;
```

7.3 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;
                …
                …
```

8 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.

8.1 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.

8.1.1 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)
...
```

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

8.1.2 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.

8.1.3 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.

8.2 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.

8.3 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
```

8.4 Dereferencing

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

Good

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

Bad

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

8.5 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*

8.6 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*
- Underscores should only be used in multi-word uppercase names where CamelCaps cannot be used e.g. *MCU.PINMODE_OD0* not *MCU.PINMODEOD0*
- Names should not be unnecessarily long nor unnecessarily abbreviated, e.g. *maxStep* not *maximumForLoopStep*, *nextPage* not *nxtpg* etc.

8.7 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(l, t, r, b);
a := i * 8 + j - m[i, j];
```

Bad

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

8.8 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;
```

8.9 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
```

8.10 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)