CFB Software

Astrobe

Oberon Programming Guide

This document 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 Programming Guide

Table of Contents

1 Introduction

The Oberon compilers included in Astrobe implement the Oberon language as defined in the latest report titled:

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

This document is generally applicable to the following editions of Astrobe:

- Astrobe for Cortex-M0, Cortex-M3, Cortex-M4 and Cortex-M7
- Astrobe for RP2040 and RP2350

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.RCCAHB1ENRGPIOD
MCU.SIOGPIOOESET
```

can be defined as:

```
MCU.RCC_AHB1ENR_GPIOD
MCU.SIO_GPIO_OE_SET
```

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 BOOLEAN

ORD(FALSE) = 0, ORD(TRUE) = 1.

3.3 CHAR

The characters of the Latin-1 set.

3.4 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

3.5 REAL

The range of valid REAL numbers is:

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

3.6 SET

The sets of integers between 0 and 31.

4 Language Extensions

4.1 ARRAY of BYTE Type Casting

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 *)
```

The converse situation is also catered for i.e. a procedure with a formal parameter of any type can accept an actual parameter which is an array of bytes of the same size.

These extensions are designed to simplify and optimise the code required for tasks such as de-serialising / serialising complex data structures for Input / Output operations, writing generic debugging trace functions etc. Examples of their use can be seen in *ByteArrays.mod* in the *General* examples supplied with Astrobe.

The following compiler warnings are generated:

```
Warning: type cast to byte array
Warning: type cast from byte array
```

NOTE: You can use SYSTEM.VAL for type casting if you would prefer not to use an Oberon Language extension. Examples of its use can be seen in *VALByteArrays.mod* in the *General* examples supplied with Astrobe.

NOTE: Compiler warnings are not generated for SYSTEM.VAL. The presence of SYSTEM in the IMPORT list warns of potentially unsafe operations.

4.2 Built-in Procedures

4.2.1 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
```

4.2.2 BITS

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

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

4.2.3 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 *)
```

4.2.4 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);
```

4.2.5 CLEAR

```
PROCEDURE CLEAR(VAR v: <record or array type>);
```

CLEAR initialises every word in a record or an array variable v to zero. It is primarily designed to be used to simplify and optimise the code required to initialise complex data structures. An example of its use can be seen in the procedure *InitHeader* in the *HCFiler* library module *HCDir*.

4.2.6 DISPOSE

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

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

4.2.7 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}

4.2.8 ORD

```
PROCEDURE ORD(x): INTEGER;
```

ORD returns the ordinal number of x which can be a BYTE, CHAR, BOOLEAN or SET expression.

4.3 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 declaration.

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

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

- Array index checks are suppressed

NOTE: The following features are not implemented for target microcontrollers with a limited number of generally accessible registers i.e. Cortex-M0 and RP2040.

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

Limitations of leaf procedures are:

- Procedures (other than standard built-in procedures and SYSTEM procedures) cannot be called from a leaf procedure
- REAL operations are restricted for systems that do not have a hardware FPU.
- 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*. 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 *)
```

4.4 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**.**

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

The *IRQTimer.TimerHandler* procedure is used to handle the interrupts and *IRQTimer.Init* contains the code required to install this handler using the *Traps.Assign* procedure.

5 Clarifications and Restrictions

5.1 Constant declarations

REAL expressions and BOOLEAN expressions are not allowed in CONST declarations.

5.2 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, assign it to a local variable to avoid the overhead of recalculating it for every iteration of the loop . For example:

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

5.3 Exports

- Anonymous record and array variables can be exported.
- String constants cannot be exported.

  NOTE: String constants can be assigned to exported variables which are read-only in modules that import them.

5.4 Pointer Types

- Pointer types can only point to named record types e.g.

  Good

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

  Bad

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

- Pointer types can only point to global record types.
- A warning is reported at compile-time when a pointer type, whose base type is a private record, is exported.

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

  This is referred to as an *opaque pointer*.

- An error is reported at compile time if a type test is performed on an imported opaque pointer variable.

- An error is reported at compile time if an imported opaque pointer variable is passed as parameter to the built-in function *NEW.*

5.5 Record Extensions

All identifiers declared in a record extension must be different from the identifiers declared in its base type record(s).

Good

```
TYPE
  R = RECORD a: INTEGER END;
  R0 = RECORD (R) b: INTEGER END;
  R1 = RECORD (R) b: REAL END;
```

Bad

```
TYPE
  R0 = RECORD a, b: INTEGER END;
  R1 = RECORD (R0) b: REAL END;
```

An exception to this is that an identifier declared in the extended record may be the same as the identifier of a *private* field declared in its *imported* base type record(s).

Good

```
MODULE M;
TYPE
  R0* = RECORD a*, b: INTEGER END;

MODULE M1;
IMPORT M;
TYPE
  R1 = RECORD (M.R0) b: REAL END;
```

5.6 PACK(x, e)

Prerequisite: 1.0 <= x < 2.0

5.7 UNPK(x, e)

Prerequisite: x >= 0

6 Implementation Size Limits

Items	Maximum
Type extension levels (configurable)	8 (Default = 4)
Types exported from a module	17
Number of modules imported*	64
Number of modules in an application	256
Number of procedures in a module	512
Number of entries in a module*	256
Maximum code size of a module	~100KB
Maximum number of parameters to a procedure*	11 (8 for Cortex-M0, RP2040)
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	64KB (32KB for Cortex-M0, RP2040)
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

6.1 Number of modules imported

The number of modules imported includes not only the modules explicitly imported in the IMPORT list but also any other modules that are implicitly imported because they are referenced in the interfaces of the direct imports. If the limit is exceeded and it is not feasible to reduce the number of imports, the solution is to subdivide the module into two or more submodules, each with fewer imports.

6.2 Number of entries in a module

Items (variables, procedures etc.) exported by a module, whose actual runtime addresses are established by the linker, are included in a section of the object module known as entries.

Note, for example, that only 1 entry is needed for the following declaration as only one absolute address is involved:

```
Ptr* = POINTER TO PtrDesc;
Ptr1* = POINTER TO PtrDesc;
PtrDesc* = RECORD i1*: INTEGER END;
```

The number of entries is reported in the compilation summary to warn if the limit is close and it might be advisable to refactor the module.

If the limit is exceeded the compilation terminates with the error message *too many entries*.

6.3 Maximum Number of parameters to a procedure

The compiler uses the processor's registers to pass parameters efficiently. Consequently, the number quoted is the absolute maximum number of parameters that can be passed to a procedure. The limit may be less in practice as some types of parameters require more than one register to be allocated. The compiler will display an error alert, no free registers, if that is the case and the code must be simplified before it will run correctly. For example, if the parameter is a complex expression, assigning it to a local variable and passing that instead may reduce the number of registers required.

In general, it is good practice to try to minimise the number of parameters passed to a procedure. For example:

- The presence of a large number of parameters often indicates that poor structured design has resulted in the procedure performing an excessive number of tasks. Analyse the functions of the procedure to see if it can be refactored into two or more simpler procedures, each performing separate tasks requiring fewer parameters.
- If any of the parameters are related to each other, consider combining them into a structured type, such as a record or an array. This approach helps organize your data and reduces the number of individual parameters you need to pass.

7 CASE Statements

7.1 Numeric CASE Statements

In the Astrobe Oberon compiler, the numeric CASE statement has been implemented in a way that can provide better execution performance than the equivalent IF-THEN sequence at the expense of memory consumption.

Numeric CASE statements are best suited to situations where:

- The case labels are naturally bytes, integers or characters
- The case labels are relatively contiguous
- There are a more than a couple of cases and they are mutually exclusive
- 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 approximately 4 times faster than the IF-THEN

ladder solution. However, the CASE approach generates approximately 8% (6% for M0) more code.

If you do use the IF-THEN ladder it will be more efficient if the expressions that are tested first have a greater probability of being TRUE.

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", "H", "I", "O", "U", "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 both 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.
```

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

Good

```
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;
```

Bad

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

8 Conditional Compilation

In the process of testing a module it is often very useful to add diagnostic traces to monitor progress. As these diagnostics are generally only required for development they can be turned off for the release version of the application using various CONST declarations.

Conditional Compilation is an optimisation which eliminates sections of code from the executable if they are included in an IF statement controlled by a BOOLEAN value declared in a CONST declaration as FALSE. The end result uses less memory and can result in improved performance.

8.1 Example of Use

The following is an example of how the feature can be exploited. This technique is particularly useful as it does **not** require any source code to be edited to switch from the *test* version of an application to the *release* version:

1. Implement a module e.g. *Trace* which includes a constant declaration *enabled* and a number of procedures to handle diagnostic information e.g.:

```
MODULE Trace;
CONST
  enabled* = TRUE;

PROCEDURE Message*(s: ARRAY OF CHAR);
  ...
  ...
```

2. Your application could then include:

```
IMPORT Trace;
  ...
  IF Trace.enabled THEN Trace.Message("Init started") END;
```

If *Trace.enabled* is declared as FALSE the resulting linked application is identical to what it would be if the entire IF statement did not exist.

3. Implement two versions of the *Trace* module, each in a separate folder. The *release* folder has the one with *enabled* set TRUE and a full implementation of each diagnostic procedure. The *test* folder has the one with *enabled* set FALSE and an empty body for each diagnostic procedure.

4. Create two configuration files for your application: a *release* configuration with a search path that includes the *release* folder and a *test* configuration file with a search path that includes the *test* folder. Use the appropriate one when you compile and link your application.

If you then need to check whether or not your code has been optimised:

- At the module level, the disassembler listings clearly show statements for which no code has been generated.
- At the application level the map file shows which folder contained the trace module that was actually linked.

8.2 Implementation Details

- Statements contained with ELSIF and ELSE are also optimised. For example, in the following no code is generated for the highlighted statements:

```
CONST
  trace = TRUE;
  debug = FALSE;

VAR
  v1: BOOLEAN;

IF debug THEN
  S1
END;

IF v1 THEN
  S1
ELSIF debug THEN
  S2
END;

IF trace THEN
  S1
ELSE
  S2
END;

IF trace THEN
  S1
ELSIF v1 THEN
  S2
ELSE
  S3
END;

IF debug THEN
  S1
ELSIF trace THEN
  S2
ELSE
  S3
END;
```

- Nested IF statements are not optimised. For example:

```
IF trace THEN
  IF debug THEN
    S1
  END
END;
```

- The constant used in the IF statement must be a value not an expression. For example, the following will not be optimised:

```
CONST
  trace = TRUE;
  debug = FALSE;

IF debug & trace THEN
  S1
END;

IF ~trace THEN
  S1
END;
```

- Source code that is optimised out must still be valid.

  Only the generated code and data is eliminated. The compiler will report any syntax errors as usual. An item will not be reported as unused even it is only referenced in source code that has been optimised out.

9 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.

9.1 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.

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.2 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.

9.3 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
```

9.4 Loop Statements

Oberon has three different types of loop statements: FOR, WHILE and REPEAT.

- If the loop is repeated a *predetermined number* of times use a FOR loop.
- If the loop is repeated *zero or more* times and a test can be performed at the beginning of the loop use a WHILE loop.

- If the loop is repeated *one or more* times and a test can be performed at the end of the loop use a REPEAT loop.

- If the loop is repeated without ever terminating use:

```
WHILE TRUE DO
…
END
```

or:

```
REPEAT
…
UNTIL FALSE
```

9.5 Dereferencing

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

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

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

9.6 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.NVIC_ISER*

9.7 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.

9.8 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];
```

9.9 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;
```

9.10 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
```

9.11 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)