Programming Conventions

Contents

This text describes the programming guidelines and source code formatting conventions which have been used for the BlackBox Component Framework. For the documentation there exist similar conventions.

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 programs published by Oberon microsystems. 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 of the BlackBox Component Framework.

1 Essentials

The most important programming conventions all center 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.
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 Component Pascal 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 consequently. Don't allow client code to "enter" your module if it doesn't fulfill the preconditions of your module's entry points (procedures, methods). In this way, you avoid propagation of foreign errors into your own code.
Use the following numbers to be consistent with the rest of the BlackBox Component Builder:

Free	0 .. 19	use for temporary breakpoints

Preconditions	20 .. 59	validate parameters at procedure entry
Postconditions	60 .. 99	validate results at procedure end
Invariants	100 .. 120	validate intermediate states (detect local error)
Reserved	121 .. 125	reserved for future use
Not Yet Implemented	126	procedure is not yet implemented
Reserved	127	reserved for future use
Silent Trap	128	termination without trap window

• 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.

• Function procedures, i.e., 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 the don't have side effects. Returning function results and assigning OUT parameters is ok.

• Only use ELSE in CASE or WITH if input data is extensible.
ELSE clauses in CASE x OF or WITH x DO statements should only be used if x is genuinely extensible. If x is not extensible, all cases that can occur should be statically known, and can be listed in the statement.

2 Oberon

For new software, don't use the special Oberon features that are not part of Component Pascal. In particular, this is the LONGREAL type and the COPY procedure. Try to avoid super calls (use composition instead of inheritance) and procedure types (use objects and methods instead).

3 Basic types

Use the types INTEGER, REAL and CHAR as defaults. BYTE, SHORTINT, LONGINT, SHORTREAL and SHORTCHAR should only be used if there is a strong particular reason to justify this choice. The auxiliary types are there mainly for interfacing purposes, for packing of very large structures, or for computation of extremely large integers. This rule is more important for exported interfaces than for hidden implementations, of course.

4 Font attributes

Programs are written in plain text, in the default color, with the following exceptions:
• Comments are written in italics, e.g. *(* this is a comment *)*
• Exported items, except for record fields and method signatures in record declarations, are written in bold, e.g. PROCEDURE **Do***;
• Keywords which indicate non-local control flow are written in bold, i.e. **RETURN**, **EXIT**, and **HALT**
• Text parts which are currently being changed and tested may temporarily be written in a different color

5 Comments

Comments which are part of an interface description (rather than a mere implementation description) have additional asterisks, a kind of "export mark" for comments, e.g. *(** guard for xyz command **)*

6 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 Dereferencing

The dereferencing operator ^ should be left out wherever possible.

Good

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

Bad

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

8 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 use all-caps identifiers with more than one character. They should be reserved for the language.

9 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. NewObject(), IsEmpty(q)
• Only declare a record type if necessary (FooDesc for each Foo are normally not needed anymore)
• 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*.

Examples
```
PROCEDURE InitDir (dir: Directory);
PROCEDURE (p: Port) Init (unit, colors: LONGINT);
```

• Guard procedures have the same name as the guarded command, except that they have a Guard suffix appended.

Example
```
PROCEDURE PasteChar;
PROCEDURE PasteCharGuard (VAR par: Dialog.Par);
```

10 White space

• A new indentation level is realized by adding one further tabulator character
• Between lists of symbols, between actual parameters, and between operators a single space is inserted

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];
```

• There is a space between a procedure name (or its export mark) and the parameter list in a declaration, but not in a call

Good
```
PROCEDURE DrawDot* (x, y: INTEGER);
DrawDot(3, 5);
```

Bad
```
PROCEDURE DrawDot*(x, y: INTEGER);
DrawDot (3, 5);
```

• 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.
• LOOP statements are never on one line
• 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, LOOP, FOR, CASE constructs is one level further indented if it does 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;

LOOP
   S0;
   IF expr THEN EXIT END;
   S1
END;

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

TYPE View = POINTER TO RECORD (Views.ViewDesc) END;

IMPORT Views, Containers,
   TextModels, TextViews;

VAR a, b: INTEGER;
   s: TextMappers.Scanner;
```

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;
      ...
```

11 Example

```
MODULE StdExample;

   IMPORT Models, Views, Controllers;

   CONST null = 0X;

   TYPE
      View* = POINTER TO RECORD (Views.View)
         a*, b*: INTEGER
      END;

   VAR view-: View;

   PROCEDURE Calculate* (x, y: INTEGER);
      VAR area: LONGINT;

      PROCEDURE IsOdd (x: INTEGER): BOOLEAN;
      BEGIN
         RETURN ODD(x)
      END IsOdd;

   BEGIN
      area := x * y;
      IF IsOdd(area) THEN area := 1000 END
   END Calculate;

BEGIN
   Calculate(7, 9)
END StdExample.
```

12 Open Source Header

Modules published under the BlackBox Component Builder Open Source License include a header text along the lines of the following template.

```
(**
    project          = "BlackBox"
    organization     = "www.oberon.ch"
    contributors     = "Oberon microsystems"
    version          = "System/Rsrc/About"
    copyright        = "System/Rsrc/About"
    license          = "Docu/BB-License"
    purpose          = ""
    changes          = "
```

```
        issues                  = "➕🏷 🏷➕"

    **)
```