# Native Oberon: Symbol and Object File Format

P. Reali

January 3, 2001

## 1 Introduction

This document describes the object and symbol file format used since PC Native Oberon beta 8.12.2000 and in binary compatibles systems.

### Change Log

- 03.01.01 Figure: Memory Layout generated by the loader.
- 24.11.00 ObjFlag (sysflag for Objects) in the Symbol File.
- 09.11.00 Symbolfile, constant value format specified.
- 02.10.00 Some corrections to the reference section. Code written in a smaller font.
- 25.09.00 Reference section changed: WinOberon support for meta-programming added. Procedure return type support for OberonX extensions.

## 2 Notation

The EBNF (Extended Backus Naur Format) is used to describe the syntax of the symbol and object file format. The semantics of the format are specified whenever needed.

We use the following writing conventions:

| | |
|---|---|
| $\mathrm{name}_{\mathrm{s}}$ | write 0X terminated string |
| $\mathrm{name}_{\mathrm{s0}}$ | write string with 0X compression |
| $\mathrm{num}_1$ | write num as 1 byte value |
| $\mathrm{num}_2$ | write num as 2 byte value |
| $\mathrm{num}_4$ | write num as 4 byte value |
| $\mathrm{num}_{\mathrm{n}}$ | write num as compressed value |

0X compression integrates the 0X terminating the string in the last character by adding 80X to it. Exceptions to this rule are the empty string ( written as 0X ) and strings containing characters bigger than 7EX. Appendix B contains the zero compression code.

## 3 Symbol File

The symbol file implements a variation of the fine-grained object fingerprinting presentend in R. Crelier dissertation, also known as Object Model (OM). The OM allows to extend a symbol file, e.g. add symbols to a module interface, without invalidating the clients of the module.

```
SymFile     =  {modname_s0}
               [SFConst {Structure name_s0 val}]
               [SFvar {[SFobjflag flag_1] [SFreadonly] Structure name_s0}]
               [SFlproc {Structure name_s0 ParList}]
               [SFxproc {[SFobjflag flag_1] Structure name_s0 ParList}]
               [SFoperator {Structure name_s0 ParList}]
               [SFalias {Structure name_s0}]
               [SFtyp {Structure}]
               SFEnd

ParList     =   {[SFvar] Structure name_s0} SFEnd
Structure   =   Basic | UserStr | oldstr_n | modno_n (name_s0 | 0X oldimpstrn_n)
Basic       =   SFtypBool ..  SFtypNilTyp
UserStr     =   [SFinvisible][SFsysflag flag_n] UserStr2
UserStr2    =   (SFtypOpenArr | SFtypDynArr) Structure name_s0
            |  SFtypArray Structure name_s0 size_n
            |  SFtypPointer Structure name_s0
            |  SFtypProcTyp Structure name_s0 ParList
            |  SFtypRecord Structure name_s0 prio_n flags_n RecStr
RecDef      =   { FieldDef }[SFtproc {MethodDef}] SFend
FieldDef    =   [SFreadonly] Structure name_s0
MethodDef   =   Structure name_s0 ParList
```

- records: invisible fields and methods are exported with name ”” (empty string)

- internal structure numbering: the first time an UserStr is exported, it is assigned a number (starting from 0, decreasing) which will be used as ”oldstr”-reference for further export

- external structure numbering: the first time an imported structure is re-exported, it is assigned a number (starting form 0, ascending) which will be used as ”oldimpstr”-reference for further export. Every imported module has an own re-export numbering

- Constant values: chars and integers are written in compressed number format.

**Basic Types Encoding**

| | | |
|---|---|---|
| SFtypBool | = | 01X |
| SFtypChar | = | 02X |
| SFtypSInt | = | 03X |
| SFtypInt | = | 04X |
| SFtypLInt | = | 05X |
| SFtypHInt | = | 06X |
| SFtypReal | = | 07X |
| SFtypLReal | = | 08X |
| SFtypSet | = | 09X |
| SFtypString | = | 0AX |
| SFtypNoTyp | = | 0BX |
| SFtypNilTyp | = | 0CX |
| SFtypByte | = | 0DX |
| SFtypSptr | = | 0EX |

**Composed Types**

| | | |
|---|---|---|
| SFtypOpenArr | = | 2EX |
| SFtypDynArr | = | 2FX |
| SFtypArray | = | 30X |
| SFtypPointer | = | 31X |
| SFtypRecord | = | 32X |
| SFtypProcTyp | = | 33X |

**Flags**

| | | |
|---|---|---|
| SFsysflag | = | 34X |
| SFinvisible | = | 35X |
| SFreadonly | = | 36X |
| SFobjflag | = | 37X |

**Section Delimiters**

| | | |
|---|---|---|
| SFconst | = | 37X |
| SFvar | = | 38X |
| SFlproc | = | 39X |
| SFxproc | = | 3AX |
| SFoperator | = | 3BX |
| SFtproc | = | 3CX |
| SFalias | = | 3DX |
| SFtyp | = | 3EX |
| SFend | = | 3FX |

## 4 Object File

The Object File is loaded in memory as in figure 4. The static data (constants) are loaded in the const section (with negative offset relative to sb), while the space for local variables is allocated in the var section (with positive offset relative to sb) and initialised to 0.

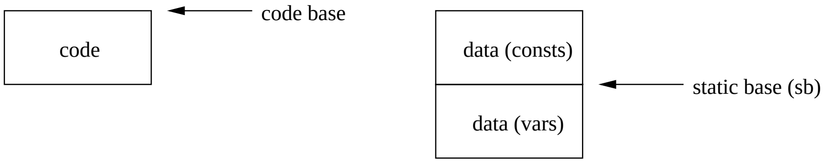

Figure 1: Memory Layout

| | | |
|---|---|---|
| ObjectFile | = | OFTag OFVersion symfilesize$_n$ SymbolFile<br>Header Entries Commands Pointers Imports VarConsLinks Links<br>Consts Exports Code Use Types References |
| OFTag | = | 0BBX |
| OFVersion | = | 0AFX |

## 4.1 Heading

Header = $\texttt{refSize}_4$ $\texttt{nofEntries}_2$ $\texttt{nofCommands}_2$ $\texttt{nofPointers}_2$
$\texttt{nofTypes}_2$ $\texttt{nofImports}_2$ $\texttt{nofVarConsLinks}_2$ $\texttt{nofLinks}_2$
$\texttt{dataSize}_4$ $\texttt{constSize}_2$ $\texttt{codeSize}_2$ $\texttt{moduleName}_\texttt{s}$

This sections gives the number of entries in the following sections.

## 4.2 Entry Section

The entry table contains the address relative to the code base of the exported procedures. Also in this table are the procedures that are assigned to a procedure variable, because the assignment requires the absolute address of the procedure.

Entries = $\texttt{82X}\ \{\texttt{entryOffset}_2\}^{\texttt{nofEntries}}$

## 4.3 Command Section

Exported procedures without parameters are commands and can be invoked by the system. *cmdOffset* is relative to the code base.

Commands = $\texttt{83X}\ \{\texttt{cmdName}_\texttt{s}\ \texttt{cmdOffset}_2\ \}^{\texttt{nofCommands}}$

## 4.4 Pointer Section

This section lists the pointers in the global variables. This information is used as root set for the current module by the garbage collector. The *pointerOffset* is relative to the static base of the module and is always a negative number (variables are stored below the static base).

Pointers = $\texttt{84X}\ \{\texttt{pointerOffset}_4\ \}^{\texttt{nofPointers}}$

## 4.5 Import Section

This section lists the modules needed by the current modules. These modules must be loaded before the current module.

Imports = $\texttt{85X}\ \{\texttt{moduleName}_\texttt{s}\}^{\texttt{nofImports}}$

## 4.6 VarConstLink Section

This section contains the fixup lists for global variables and constants (including type descriptors). The list contains the *count* of fixes to be done and their *offset* relative to the code base. The address of the entry has to be added to the value found at the offset!

All the entries of the current module are grouped into the list with *mod* = 00 and *entry* = $0FFFFH$.

For every imported entry, *mod* is the module where the entry is defined (as implicitly numbered in the import section), *entry* is always 0. The Use Section (4.10) contains the table that maps an imported symbol to a fixup list.

VarConstLinks = `8DX` $\{$ `VarConstLinkEntry` $\}^{\mathtt{nofVarConsLinks}}$
VarConstLinkEntry = $\mathtt{mod}_1$ $\mathtt{entry}_2$ $\mathtt{count}_2$ $\{\mathtt{offset}_2\}^{\mathtt{count}}$

## 4.7 Link Section

This section contains the fixup list for procedure calls, system calls, and some other special fixups in the code. *offset* is relative to the code base. That location contains the address of the next fixup (this fixup chain is embedded in the code).

The following *mod / entry* have special meanings:

| | |
|---|---|
| 00 / 255 | case table fixup |
| 00 / 254 | local procedure assignment |
| 00 / 253 | system call to NewRec |
| 00 / 252 | system call to NewSys |
| 00 / 251 | system call to NewArr |
| 00 / 250 | system call to Start |
| 00 / 249 | system call to Passivate |
| 00 / 247 | system call to Lock |
| 00 / 246 | system call to Unlock |

Links = `86X` $\{\mathtt{LinkEntry}\}^{\mathtt{nofLinks}}$
LinkEntry = $\mathtt{mod}_1$ $\mathtt{entry}_1$ $\mathtt{offset}_2$

## 4.8 Const and Code Sections

The *Consts* are loaded in memory beginning at the static base.

Consts = `87X` $\{\mathtt{char}_1\}^{\mathtt{constSize}}$
Code = `89X` $\{\mathtt{char}_1\}^{\mathtt{codeSize}}$

## 4.9 Export Section

This section lists the exported symbols of the module

Exports = `88X` $\mathtt{nofExports}_2$ $\{\mathtt{ExportEntry}\}^{\mathtt{nofExports}}$ `0X`
ExportEntry = $\mathtt{FP}_n$ $\mathtt{fixup}_n$ `[1X ExportRecord]`
ExportRecord = $\mathtt{oldref}_n$
| $\mathtt{tdentry}_n$ `[1X ExportRecord]` $\mathtt{nofFPs}_2$ $\{\mathtt{FP}_n$ `[1X ExportRecord]`$\}^{\mathtt{nofFPs}}$ `0X`

The Export Section contains the information and linking point of all the exported symbols in the module. The *ExportEntry* describes different entry kinds, depending on the value of the *fixup*:

- $fixup < 0$ Fixup is the offset of a variable relative to the static base. If the variable type is a record, *ExportRecord* will describe it.
- $fixup = 0$ Anchor for a named record type, always followed by *ExportRecord*.
- $fixup > 0$ Fixup is the offset of a procedure entry point relative to the code base; it never has an ExportRecord.

A record may be described in two ways (recognized by the first value read):

- *tdentry* > 0 tdentry is the offset of the pointer to the type descriptor in the constant section; followed by the fingerprints of the base type (if existent), the public and private record fingerprints, the methods and fields fingerprints. If a record field has record type, then it is followed by the description of the type.
- *oldref* < 0 the -oldref-th explicitely described type descriptor.

## 4.10 Use Section

| | | |
|---|---|---|
| Use | = | 08AX {UsedModules} 0X |
| UsedModules | = | moduleName$_{s0}$ {UsedVar \|UsedProc \|UsedType} 0X |
| UsedVar | = | FP$_{n}$ varName$_{s0}$ fixlist$_{n}$ [1X UsedRecord] |
| UsedProc | = | FP$_{n}$ procName$_{s0}$ offset$_{n}$ |
| UsedType | = | FP$_{n}$ typeName$_{s0}$ 0X [1X UsedRecord] |
| UsedRecord | = | tdentry$_{n}$ [FP$_{n}$ "@"] 0X |

The Use Section contains the information for the linker about all the imported entries (Variables, Procedures and Type Descriptors). Every entry has the format `fingerprint name value`. Variables have a positive value, procedures a negative one and types have $value = 0$.

**Variables** *fixlist* is an index to a fixup list in the VarConstLink Section (4.6. The address to be patched is found in the Export Section (4.9) of the module exporting the variable using the fingerprint.

**Procedures** The fixup chain for this call starts at $-offset$ relative to the code base. The address to be patched is found in the Export Section (4.9) of the module exporting the procedure.

**Type Descriptors** For every imported type descriptor, an hidden copy of the pointer to the descriptor is allocated in the local constant section at offset *tdentry.* The address to be patched is found using the fingerprint in the Export Section (4.9) of the module exporting the Type.

## 4.11 Types Section

This section contains the description about the type descriptors.

| | | |
|---|---|---|
| Types | = | 8BX {TypeEntry}$^{\mathtt{nofTypes}}$ |
| TypeEntry | = | size$_{4}$ tdaddr$_{2}$ Base Count name$_{s}$ Methods Pointers |
| Base | = | module$_{2}$ entry$_{4}$ |
| Count | = | nofMethods$_{2}$ nofInheritedMethods$_{2}$ nofNewMethods$_{2}$ nofPointers$_{2}$ |
| Methods | = | {methodNumber$_{2}$ entryNumber$_{2}$}$^{\mathtt{nofNewMethods}}$ |
| Pointers | = | {pointerOffset$_{4}$}$^{\mathtt{nofPointers}}$ |

There are 3 different *Base* allowed:

**no base** module and entry are equal to -1

**local record** module equal to 0; entry is the offset in the constant section of the pointer to the base type

**imported record** module is an index in the table of imported modules; entry is the fingerprint of the record (to be found and checked in the export section of the module)[1]

[1] In fact this is not really needed, because for every imported record, a pointer to it is created in the local constants section. The module number could be safely ignored and entry be the offset in the constants; the fingerprint would be checked in the use section when fixing the reference to the td.

### 4.12 Reference Section

The reference section is used by the Oberon trap handler to display the values on the stack when the trap occured. It may be also used as a support for meta-programming.

```
Reference   =  8CX BodyRef {ProcRef}
BodyRef     =  0F8X offset_n name_s {Variable}
ProcRef     =  0F9X offset_n nofPars_1 RetType_1 procLev_1 slFlag_1 name_s {Variable}
RetType     =   0X | VarType | ArrayType | Record
ArrayType   =  12X | 14X | 15X. (* static, dynamic, open *)
Record      =  16X
Variable    =  VarMode ( VarType | ArrayVar | RecordVar ) offset_n name_s
VarMode     =  1X | 3X. (* direct, indirect *)
RecordVar   =  (16X|1DX) tdadr_n (* record, recordpointer *)
            =
ArrayVar    =  81X..8EX dim_n
VarType     =  1X ..0FX
               (*byte, boolean, char, shortint, integer, longint
                real, longreal, set, , , , ptr, proc, string *)
```

The name of a module body is always "$$".

*ProcRef* gives an extended description of the procedure: procLev is the procedure level, slFlag = 1 indicates that the static link is also pushed. RetType = 0 if the procedure returns no value.

*ArrayVar*: for open arrays dims = 0.

*RecordVar*: tdadr is an offset relative to the module static base.

# A Oberon Kernel System Calls

This appendix has been contribued by Pieter Muller. Many thanks.

```
TYPE Object = POINTER TO RECORD END;
TYPE Body = PROCEDURE (typetag: LONGINT; self: Object);
PROCEDURE CreateActivity(body: Body; priority: LONGINT; flags: SET; obj: Object);
```

Create a thread associated with the active object `obj`. `body` contains the body method of the active object. `priority` and `flags` are the values specified in the annotation of the body. This call is generated by the compiler when an active object (a `POINTER TO RECORD` variable with a `BEGIN-END` body) is allocated with `NEW`. First the object is allocated as usual, then its initializer is called (if defined), and then `Create` is called to activate it.

The call creates a new thread that has the body entry point as initial instruction pointer. A stack is set up for the local variables of the thread, in such a way so that return from the body will terminate the thread. A stack overflow will cause an extension of the stack, or a trap if no more memory is available to the thread. Any trap will cause the thread to be either restarted at the body, or terminated (this depends on `flags`). It is not (yet) defined what happens to the locks that are held by a trapping thread. Priority levels are not yet defined. The thread of an active object will anchor the object until it terminates. After that the object may remain anchored by other references to it, but it can not become active again. It can remain in the system as a protected object.

```
PROCEDURE Lock(obj: Object; exclusive: BOOLEAN);
```

Lock protected object `obj`. The compiler generates this call at the entry to a method with the `EXCLUSIVE` or `SHARED` annotation (called a protected method). `exclusive` indicates which is the relevant case. Only one thread can lock an object exclusively, and many threads may obtain a shared lock when no exclusive lock is held. A thread is not allowed to re-enter its own exclusive region. Any object with exclusive or shared methods (also if only in an extension) can be locked. The compiler must indicate this in the type descriptor of the object, so that the object can be allocated with the relevant header. (As an approximation, any object with methods may be treated as a protected object). Shared locks may be implemented identical to exclusive locks in the simplest case.

```
PROCEDURE Unlock(obj: Object; dummy: LONGINT);
```

Unlock protected object `obj`. The compiler generates this call at the exit of a protected method. The relevant lock is released. (The `dummy` parameter is a placeholder to be used or removed later).

```
TYPE Condition = PROCEDURE (slink: LONGINT): BOOLEAN;
PROCEDURE Passivate(cond: Condition; slink: LONGINT; obj: Object; flags: SET);
```

Passivate the current thread until some condition becomes true. The compiler generates this call for the `PASSIVATE` statement. The boolean condition is compiled in a separate procedure which is logically nested in the scope where the passivate resides, and which returns the boolean result of the expression. The static link value to that scope is passed in the `slink` parameter. This value is used when calling the condition procedure, so that it can access the variables of the enclosing scope. `obj` points to the object instance containing the passivate statement. Bit 0 of the `flags` parameter is set if the compiler detects a ‘global’ condition, i.e. a boolean expression with function calls or reference to non-local variables.

```
PROCEDURE NewRec*(VAR p: SYSTEM.PTR; typetag: LONGINT);
```

This call is generated for the `NEW` procedure with a `POINTER TO RECORD` parameter. `typetag` is a the address of a type descriptor for the specified record type. From the type descriptor can be learned if the relevant object is a protected object, in which case a heap block with the required protected object header is allocated. (It would be advantageous to have the compiler generate a separate kernel call for this case). `p` returns the allocated pointer value.

```
PROCEDURE NewArr*(VAR p: SYSTEM.PTR; elemTag, numElems, numDims: LONGINT);
```

This call is generated for the `NEW` procedure with a `POINTER TO ARRAY OF` parameter, where the array elements are pointers or records containing pointers. `elemTag` is the address of a type descriptor for the element record type, or 0 in the case of an array of pointers. `numElems` and `numDims` indicate the total size and number of dimensions of the array. The array is allocated with a special header where the sizes of the different dimensions are stored. These fields are initialized by code generated by the compiler after the kernel call.

```
PROCEDURE NewSys*(VAR p: SYSTEM.PTR; size: LONGINT);
```

This call is generated for the `SYSTEM.NEW` procedure to allocate a block of memory that does not contain any pointers that have to be traced by the garbage collector. It is also used for the `NEW` procedure with a `POINTER TO ARRAY OF` parameter, where the array elements do not contain pointers.

Kernel call numbers:

| | |
|---|---|
| 244 | - |
| 245 | - |
| 246 | Unlock |
| 247 | Lock |
| 248 | - |
| 249 | Passivate |
| 250 | CreateActivity |
| 251 | NewArr |
| 252 | NewSys |
| 253 | NewRec |

# B  Zero Compression of strings

```
PROCEDURE WriteString(VAR R: Files.Rider; VAR s: ARRAY OF CHAR);
  VAR i: INTEGER; ch: CHAR;
BEGIN
  i:=0; ch:=s[i];
  IF  ch=0X  THEN  Files.Write(R, 0X); RETURN  END;
  WHILE  (ch#0X) & (ch<7FX)  DO  INC(i); ch:=s[i]  END;
  IF  i>1  THEN  Files.WriteBytes(R, s, i-1)  END;
  IF  ch=0X  THEN  Files.Write(R, CHR(ORD(s[i-1])+80H))
  ELSE
    IF  i>0  THEN  Files.Write(R, s[i-1])  END;
    Files.Write(R, 7FX);
    REPEAT  Files.Write(R, ch); INC(i); ch:=s[i]  UNTIL ch=0X;
    Files.Write(R, 0X)
  END
END WString;

PROCEDURE ReadString(VAR R: Files.Rider; VAR s: ARRAY OF CHAR);
  VAR i: INTEGER; ch: CHAR;
BEGIN i := 0;
  LOOP Files.Read(R, ch);
    IF ch = 0X THEN s[i] := 0X; RETURN
    ELSIF ch < 7FX THEN s[i]:=ch; INC(i)
    ELSIF ch > 7FX THEN
      s[i] := CHR(ORD(ch)-80H); s[i+1] := 0X; RETURN
    ELSE (* ch = 7FX *) EXIT END
  END;
  LOOP Files.Read(R, ch);
    IF ch = 0X THEN s[i]:=0X; RETURN
    ELSE s[i]:=ch; INC(i) END
  END
END ReadString;
```