Aos/Bluebottle: Symbol and Object File Format

P. Reali/U. Glavitsch

Nov. 17, 2006

Introduction

This document describes the object and symbol file format used in Aos/Bluebottle for releases not older than Nov. 17, 2006.

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:

name	write 0X terminated string
num 1	write num as 1 byte value
num 4	write num as 4 byte value
num	write num as compressed value

The main idea is to write all integer numbers as compressed values. There are, however, a few places where integers are written as fixed size numbers. For instance, it often occurs that the space for an integer is reserved at a certain position in the object file and the actual value is written later. In these cases, four bytes are reserved for that integer independently of the actual value. At some other locations, one byte integers are written.

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}
            [SFConst {Structure name val}]
            [SFvar {[SFreadonly] Structure name}]
            [SFxproc {Structure name ParList}]
            [SFoperator {Structure name ParList}]
            [SFcproc {Structure name ParList code}]
            [SFalias {Structure name}]
            [SFtyp {Structure}]
            SFEnd
```

```
ParList    = {[SFvar] Structure name} SFEnd
Structure  = Basic | UserStr | oldstr | modno (name | 0X oldimpstrn)
Basic      = SFtypBool .. SFtypNilTyp
UserStr    = [SFinvisible][SFsysflag flag] UserStr2
UserStr2   = (SFtypOpenArr | SFtypDynArr) Structure name
           | SFtypArray Structure name size
           | SFtypPointer Structure name
           | SFtypProcTyp Structure name ParList
           | SFtypRecord Structure name prio flags RecStr
RecDef     = { FieldDef }[SFtproc {MethodDef}] SFend
FieldDef   = [SFreadonly] Structure name
MethodDef  = Structure name 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

Basic Types Encoding

SFtypBool	= 1
SFtypChar8	= 2
SFtypChar16	= 3
SFtypChar32	= 4
SFtypInt8	= 5
SFtypInt16	= 6
SFtypInt32	= 7
SFtypInt64	= 8
SFtypFloat32	= 9
SFtypFloat64	= 10

SFtypSet	= 11
SFtypString	= 12
SFtypNoTyp	= 13
SFtypNilTyp	= 14
SFtypByte	= 15
SFtypSptr	= 16
SFmod1	= 17

Composed Types

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

Flags

SFsysflag	= 34X
SFinvisible	= 35X
SFreadonly	= 36X

Section Delimiters

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

Object File

ObjectFile	=	`OFTag NoZeroCompression OFVersion symfilesize SymbolFile` `Header Entries Commands Pointers Imports VarConsLinks Links` `Consts Exports Code Use Types ExceptionTable PtrsInProcBlock References`
OFTag	=	`0BBX`
NoZeroCompression	=	`0ADX`
OFVersion	=	`0B1X`

Heading

Header = `refSize 4 nofEntries 4 nofCommands 4 nofPointers 4`

`nofTypes 4 nofImports 4 nofVarConsLinks 4 nofLinks 4`

`dataSize 4 constSize 4 codeSize 4 exTableLen 4`

`nofProcs 4 maxPtrs 4 moduleName`

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

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 = `82X { entryOffset }` $^{\text{nofEntries}}$

Command Section

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

Commands = `83X { cmdName cmdOffset }` $^{\text{nofCommands}}$

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 = `84X { pointerOffset }` $^{\texttt{nofPointers}}$

Import Section

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

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

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!

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 contains the table that maps an imported symbol to a fixup list.

VarConstLinks = `8DX { VarConstLinkEntry }` $^{\texttt{noVarConsLinks}}$

VarConstLinkEntry = `mod 1 entry count 4 { offset }` $^{\texttt{count}}$

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 { LinkEntry } nofLinks  { FixupCount } nofEntries  caseTableSize
LinkEntry  = mod 1 entry 1 offset
FixupCount = count
```

Const and Code Sections

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

```
Consts = 87X {char 1}
Code   = 89X {char 1}
```

Export Section

This section lists the exported symbols of the module

```
Exports      = 88X nofExports 4 {ExportEntry} 0X
ExportEntry  = FP fixup [1X ExportRecord]
ExportRecord = oldref
             | tdentry [1X ExportRecord] nofFPs 4 {FP [1X ExportRecord]} 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.

Use Section

Use = `08AX {UsedModules} 0X`

UsedModules = `moduleName {UsedVar |UsedProc |UsedType} 0X`

UsedVar = `FP varName fixlist [1X UsedRecord]`

UsedProc = `FP procName offset`

UsedType = `FP typeName 0X [1X UsedRecord]`

UsedRecord = `tdentry [FP "@"] 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. The address to be patched is found in the Export Section 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 of the module exporting the procedure.

Type Descriptors

For every imported type descriptor, a 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 of the module exporting the Type.

Types Section

This section contains the description about the type descriptors.

Types = `08BX {TypeEntry}` ^{nofTypes}

TypeEntry = `size tdaddr Base Count name Methods Pointers`

Base = `module entry`

Count = `nofMethods nofInheritedMethods nofNewMethods nofPointers 4`

Methods = `{ methodNumber entryNumber }` ^{nofNewMethods}

Pointers = `{ pointerOffset }` ^{nofPointers}

There are 3 different *Base* allowed:

no base
: module and entry are equal to -1

local record
: module equal to -1; 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]

Exception Table Section

This section contains the description about the exception table.

ExceptionTable = `08EX { ExTableEntry }` ^{exTableLen}

ExTableEntry = `0FEX pcFrom pcTo pcHandler`

PtrsInProcs Section

This section contains the description about the pointer locations on the stack for each procedure. A procedure can be a module procedure, a method or the body of either a module or an object.

PtrsInProcs = `08FX { ProcEntry }` ^{nofProcs}

ProcEntry = `codeoffset beginoffset endoffset nofPtrs 4 { pointer }` ^{nofPtrs}

The maximum of the number of pointers over all procedures is stored in the variable maxPtrs in the Header Section.

Reference Section

The reference section is used by the Oberon trap handler to display the values on the stack when the trap occured. With some extensions (i.e. object types) it may be also used for some meta-programming.

Reference = `8CX {ProcRef}`

ProcRef = `0F8X offset name {VarMode VarType [dim] [td] offset name}`

VarMode	=	`Direct | Indirect`
VarType	=	`Byte | Bool | Char | SInt | Int | LInt | Real | LReal | Set | Pointer | Proc | String`
Direct	=	`1X`
Indirect	=	`3X`
Byte	=	`1X | 81X`
Bool	=	`2X | 82X`
Char	=	`3X | 83X`
SInt	=	`4X | 84X`
Int	=	`5X | 85X`
LInt	=	`6X | 86X`
Real	=	`7X | 87X`
LReal	=	`8X | 88X`
Set	=	`9X | 89X`
Pointer	=	`0DX | 8DX`
Proc	=	`0EX | 8EX`
String	=	`0FX`

A VarType ≥ 80X means an array of the given type; in this case the number of dimensions must follow the type. Open Arrays have dimension 0.

Oberon Kernel System Calls

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

```
TYPE ProtectedObject = POINTER TO RECORD END;  (* protected object (10000)
*)
TYPE Body = PROCEDURE (typetag: LONGINT; self: ProtectedObject);
PROCEDURE CreateActivity(body: Body; priority: LONGINT; flags: SET; obj:
ProtectedObject);
```

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: ProtectedObject; 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: ProtectedObject; 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: ProtectedObject;
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

Footnotes

... module)[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.

---

ulrike.glavitsch@inf.ethz.ch