Electron Generic Binary Interface Specification

Version 0.9.0
Madd Games
September 2016

Table of Contents

1. Introduction

This document defines the *Electron Generic Binary Interface* (*EGBI*) – the architecture-independent and OS-independent set of assumptions that an Electron compiler and runtime library should enforce. Supplementary documents fill in gaps in this document in reference to a specific OS and architecture.

The purpose of the EGBI is to define an object-oriented binary interface which allows for the constructions of portable libraries and applications in the Electron programming language, or any other programming language that wishes to follow this standard. The objectives can be outlined as follows:

- Class code produced by one compiler shall be compatible with class code produced by another compiler – both classes must be able to instantiate each other, call each other's methods, etc.
- If a code change in one class allows other class code to be compiled without change, it must also not require recompilation of classes which already use it.
- An object-oriented, and as highly performing as possible, system must be provided.

1.1. Data Types and Structures

Structures defined in this document will be presented in the C programming language's format, and normal alignment shall always be used. In describing those structures, fixed-width C integer types (`uintX_t` / `intX_t`) shall be used, alongside the following types defined by this specification:

- `natural_t` is a signed integer with the target platform's natural width; that is, `int32_t` for 32-bit platforms, `int64_t` for 64-bit platforms, etc.
- `unatural_t` is an unsigned integer with the target platform's natural width; that is, `uint32_t` for 32-bit platforms, `uint64_t` for 64-bit platforms, etc.

1.2. Path Mangling

Electron paths (e.g. full class names such as "`elec.rt.System`") contain dots (`.`) and hence cannot be used as symbol names in object files. When such a path is needed in a symbol name, it shall be mangled according to the *path mangling procedure*. A mangled name is denoted by `#(`*`name`*`)`, for example `#(elec.rt.System)`. The procedure works by splitting the path into tokens delimited by '.', prefixing each one with a decimal number indicating its length, then merging them together; for example, `#(elec.rt.System)` is `4elec2rt6System`.

2. Type Definitions

A compiled class or interface is described by a data structure called a *type definition*, located at the symbol `_Elec_TypeDef_#(`*`name`*`)`, where *name* is the full name (path) of the class, for example `elec.rt.String`. The structure of the definition is called `struct elec_typedef` and is defined as follows:

```
struct elec_typedef
{
     unatural_t              tdFlagsAndDataSize;
     struct elec_typedef*    tdParentClass;
     void*                   tdInstanceInit;
     unatural_t              tdNumInterfaces;
     struct elec_typedef**   tdInterfaces;
     unatural_t              tdSize;
     const char*             tdName;
     void*                   tdInstanceFini;
};
```

One of the *flags* in `tdFlagsAndDataSize` decides whether this is a class or an interface. The meanings of some fields is different for a class and for an interface.

2.1. Flags

The *flags* are placed in the top 8 bits of `tdFlagsAndDataSize`, regardless of its size. The bits have the following meanings:

- Bits 0-3 are reserved and must be clear.
- Bit 4 (`TD_INTERFACE`) – if set, this is an *interface*; if clear, this is a *class*.
- Bit 5 (`TD_FINAL`) – if set, this class or interface is final, and cannot be extended.
- Bit 6 (`TD_ABSTRACT`) – if set, this class is abstract, and cannot be instantiated (only its children can). *Must be zero if this is an interface*.
- Bit 7 is reserved and must be zero.

2.2. Class Definition

For a *class*, the fields have the following meanings:

- `tdFlagsAndDataSize` contains *flags* in its upper 8 bits; the rest is the *data size* – the number of bytes per instance of this type that is used by code in this class.
- `tdParentClass` points to the type definition of the parent class. If no parent was declared in the source code, this must point to `_Elec_TypeDef_4elec2rt6Object`. The definition of `elec.rt.Object` itself has this field set to `NULL`.

- `tdInstanceInit` points to the *initializer function*, if any, or `NULL`. The invocation of this function is described in the section *Instantiating Classes*.
- `tdNumInterfaces` specified the number of interfaces implemented by this class, if any.
- `tdInterfaces` points to an array of pointers to type definitions specifying the interfaces implemented by this class. This is ignored if `tdNumInterfaces` is zero.
- `tdSize` is `sizeof(struct elec_typedef)`. This if for the purposes of future compatibility.
- `tdName` points to a NUL-terminated ASCII string specifying the full name (path) of this class; e.g. "`elec.rt.String`".
- `tdInstanceFini` points to the *finalizer function*, if any, or `NULL`. The invocation of this function is described in the section *Finalizing Classes*.

2.3. Interface Definition

For an *interface*, the fields have the following meanings:

- `tdFlagsAndDataSize` contains *flags* in its upper 8 bits; the rest is the size of the *data area* which contains function pointers to implemented functions.
- `tdParentClass` is set to `NULL`.
- `tdInstanceInit` is set to `NULL`.
- `tdNumInterfaces` specified the number of interfaces that this interface extends, if any.
- `tdInterfaces` points to an array of points to type definitions specifying the interfaces extended by this one. This is ignored if `tdNumInterfaces` is zero.
- `tdSize` is `sizeof(struct elec_typedef)`. This if for the purposes of future compatibility.
- `tdName` points to a NUL-terminated ASCII string specifying the full name (path) of this interface; e.g. "`elec.io.IBinaryOutputStream`".
- `tdInstanceFini` is set to `NULL`.

2.4. Field and Method Resolution

The object that defines a class or interface must also contain offsets to fields and methods within the class or interface data area. The offset into the data area to a field named *`FieldName`* is found by reading the global variable `_Elec_Field_#(`*`ClassOrInterfaceName`*`)_`*`FieldName`*, which has type `unatural_t`. The offset into the data area containing a function pointer to an implementation of a certain signature of a method is found in the global variable `_Elec_Method_#(`*`ClassOrInterfaceName`*`)_#(`*`MethodName`*`)_#(`*`Arg1TypeName`*`)_#(`*`Arg2TypeName`*`)_…`. See *Section 5. Objects* for more information.

3. Calling Convention

The exact calling convention depends on architecture and operating system. However, certain things need to be common for all implementations to create a consistent interface and minimize dependencies. This section thus explains those commonalities.

3.1. Registers

The EGBI asserts the existence of 3 registers, which shall be mapped to machine registers by supplementary ABIs:

- The *argument-register* stores a pointer to an array of arguments to the callee, located on the caller's stack frame. The callee may destroy this register.
- The *this-register* contains the value of `this` if the callee is non-static; `NULL` otherwise. It may be destroyed by the callee.
- The *count-register* contains the number of arguments. It may be destroyed by the callee.
- The *frame-info-register* stores a pointer to the caller's frame information (described below), within the caller's stack frame. The value of this register must be preserved by the callee.
- The *return-register* is used to store the return value of a function.

3.2. Frame Information

When a function is called, it sets up *frame information* on its stack frame, which is used to form a linked list that can be used to unwind the stack for exception handling. The frame information is contained in a data structure called `struct elec_frameinfo` which is passed from function to functions in the *frame-info-register*. The structure is defined as follows:

```
struct elec_frameinfo
{
    struct elec_frameinfo*          fiDown;
    struct elec_funcinfo*           fiFunc;
    char                            fiArch[];
};
```

- `fiDown` points to the frame information of the previous function (the one that called the function which this frame information describes); this is `NULL` for the deepest stack frame visible to the Electron Runtime.
- `fiFunc` points to information about the function, and is used to figure out which exceptions the function catches etc. This is described in the section *Exception Handling*.
- `fiArch` contains unknown, architecture-specific data. Refer to the architecture's Electron ABI Supplement for information on its format on the given architecture.

3.3. Caller Responsibilities

The caller shall:

1. Allocate the array of arguments on the stack, and fill it in. Each entry shall have the size of `natural_t`.
2. Load the pointer to the argument array into the *argument-register*.
3. Load the number of arguments into the *count-register*.
4. The pointer to the current frame information must be placed in the *frame-info-register*; see *Callee Responsibilities* below on how to maintain this.
5. Managed objects passed as arguments to the function belong to the callee; the callee will `decref` them.
6. Issue a call to the function (the callee) in an architecture-specific way.
7. Release the arguments from the stack.
8. The return value of the callee now belongs to you; you must `decref` it once you don't need it, if it is managed.

3.4. Callee Responsibilities

The callee shall, upon entry:

1. Allocate space on the stack for its frame information.
2. Set `fiDown` to the value of *frame-info-register* passed by the caller, `fiFunc` to the callee's function information (described in *Exception Handling*) and `fiArch` to whatever data the architecture demands.
3. Set the value of *frame-info-register* to point to the new frame information.
4. Keep the value of *frame-info-register* to call other functions.

Note that the creation of the frame information must be interrupt-safe, since some exceptions may be thrown by asynchronous signals. The architecture supplement explains how to do this on a specific architecture.

The callee shall, upon exit:

1. Deallocate all managed arguments; that is, `decref` them.
2. Restore the *frame-info-register* from `fiDown`.
3. Release anything that was pushed on the stack by the callee, such that the stack returns to the state it was left in by the caller.
4. Return to the caller in an architecture-specific way.

4. Exception Handling

In the *Electron Programming Language*, exceptions are caught by `try-catch` blocks, and thrown using the `throw` statement. An exception is a managed object which is an instance of `elec.rt.Exception` (or one of its children). At the binary level, an exception is thrown as follows:

- The pointer to the exception instance is stored in the *this-register*; it now belongs to the runtime (so the runtime is responsible for `decref`'ing it). So if the exception is thrown with the typical `throw new WhateverError(…)` statement, the reference count should be 1 at this point.
- The current frame information must be placed in the *frame-info-register*.
- Call the `_Elec_Throw` function in an architecture-specific method.

The runtime will then unwind the stack by following frame information, `decref`'ing local variables of functions that don't catch exceptions, jumping to an exception handler if found, and finally, if nobody wants to catch the exceptions, aborting. This section will explain the details.

4.1. Function Information

The *function information structure* (`struct elec_funcinfo`) is specified for each function, and contains all information necessary to unwind the stack, deallocate local variables, jump to exceptions handler, and debug. A pointer to this structure is placed in each invocation's frame information (in the `fiFunc` field). The structure is defined as follows:

```
struct elec_funcinfo
{
    unatural_t                  fnSize;
    const char*                 fnName;
    unatural_t                  fnNumVars;
    struct elec_lvinfo*         fnVars;
    unatural_t                  fnNumTry;
    struct elec_tryinfo*        fnTry;
};
```

- `fnSize` is set to `sizeof(struct elec_funcinfo)`; for future compatibility.
- `fnName` points to an ASCIIZ string indicating the human-readable full name (path) of the function.
- `fnVars` points to an array of variable descriptions necessary to release local variables of this frame (described below), and `fnNumVars` is the number of entries in this array.
- `fnTry` is an array of try-descriptors necessary to decide whether an exception is caught (described below) and `fnNumTry` is the number of entries in this array.

4.1.1. Local Variable Description

A *local variable description* is emitted for each *managed* local variable, and is needed so that the unwinding code can release our local variables. Since a local variables may be confined to scope (e.g. within a loop), that is included in the description; variables outside of the scope being unwound are not released by the unwinder, since they're assumed to already be released. The structure is defined as follows:

```
struct elec_lvinfo
{
     unatural_t                    lvScopeBegin;
     unatural_t                    lvScopeEnd;
     natural_t                     lvOffset;
};
```

- `lvScopeBegin` and `lvScopeEnd` define the scope of the variable in an architecture-specific way (they are usually memory addresses of the instruction that allocates the variable and the one that releases it, or something similar).
- `lvOffset` is the offset relative to the beginning of the frame information structure to the memory address of the variable.

When a frame does not catch an exception that is being thrown, the unwinder will release the variables according to the following pseudo-code:

```
struct elec_lvinfo *lvinfo;
struct elec_frameinfo *finfo;

if (isWithinScope(finfo, lvinfo))
{
     unatural_t addrObject = (unatural_t)finfo + lvinfo->lvOffset;
     _Elec_Object *obj = *((_Elec_Object**)addrObject);
     _Elec_DecRef(obj);
};
```

4.1.2. Try-Description

A *try-description* defines a block of code within the function which catches exceptions (the interior of a `try` construct). It also defines how to jump to the exception handler for each exception within the block. It is defined as follows:

```
struct elec_tryinfo
{
     unatural_t                    tryScopeBegin;
     unatural_t                    tryScopeEnd;
     unatural_t                    tryNumCatch;
```

```
    struct elec_catchinfo*        tryCatch;
};
```

- `tryScopeBegin` and `tryScopeEnd` define the scope of the block in an architecture-specific way.
- `tryCatch` points to an array of *catch-descrptions* specifying how to handle each exception being caught, and `tryNumCatch` is the number of entries in this array.

4.1.3. Catch Description

A *catch-description* describes an exceptions caught by a `try` block and is defined as follows:

```
struct elec_catchinfo
{
    struct elec_typedef*          catchType;
    unatural_t                    catchLand;
};
```

- `catchType` points to the class definition (see *Section 2. Type Definitions*) of the exception being caught. All its children are caught too.
- `catchLand` is the memory address of the instruction to jump to if this exception is thrown within the scope of the `try` block; this is known as the *landing pad*.

The catch list shall be sorted such that more specific exception types come first; i.e. children before parents.

When the exception is caught, the unwinder will jump to the landing pad with the following state:

- The *this-register* shall contain the exception instance; this now belongs to the landing pad code (and it must `decref` it when it's no longer needed).
- The *frame-info-register* shall contain the frame information structure belonging to the function containing the landing pad.

4.2. Uncaught Exceptions

If the unwinder gets to the bottom of the stack without the exception being caught, it shall print out the name of the exception class, the message it contains, followed by the stack trace (as obtained by the frame information), and shall abort as if by C library `abort()` function.

5. Objects

A *managed object* is created by instantiating a class. The Electron Runtime provides automatic memory management, which works by reference counting. When a class is first instantiated, the new object has a reference count of 1. Every time the object is stored in a new variable, the reference count is incremented; this operation is referred to as `incref`. Every time a variable containing a reference to the object is deallocated (falls out of scope), the reference count is decremented; this operation is referred to as `decref`. When a `decref` results in the reference count becoming zero, the object is *destroyed*. All of those operations are described in detail in this section.

5.1. Object Description

A reference to an object is represented as a pointer to its *object description*, which includes the reference count, data area for each class and interface constituting an object, etc. The object description structure is called `struct elec_object` and is defined as follows:

```
struct elec_object
{
     unatural_t                         objRefCount;
     struct elec_typedef*               objMainClass;
     unatural_t                         objNumSubs;
     struct elec_subobject*             objSubs;
};
```

- `objRefCount` is the reference count.
- `objMainClass` points to the type definition of the *main class* of this object; that is the class that was instantiated to create this object.
- `objSubs` is an array of *sub-objects* that constitute this object; each sub-object is used to contain the data area of a class in the main class's hierarchy, and of the implemented interfaces. That is, this array contains a sub-object for each ancestor of the main class, for the main class itself, and for each implemented interface, and the structure is described below. `objNumSubs` is the number of entries in the array.

Each sub-object is defined as follows:

```
struct elec_subobject
{
     struct elec_typedef*               soType;
     void*                              soData;
};
```

- `soType` points to the type definition of the class or interface whose data area is contained in this sub-object.

- `soData` points to the data area for the type, whose size shall be equal to the size defined by `soType`.

5.2. Instantiating a Class

From the binary's perspective, a class is instantiated by taking the following steps:

1. Loading the pointer to the type definition (`struct elec_typedef`) into the *this-register*.
2. Ensuring the frame information is in the *frame-info-register*.
3. Calling `_Elec_New` in an architecture-specific way.
4. The new instance, with an initial reference count of 1, is stored in the *return-register*.

The Runtime (which implements `_Elec_New`) has to parse the type definition to allow it to create the instance. In order to do this, it must first validate whether this class may be instantiated. This is done by taking the following steps:

- If the definition has the `TD_INTERFACE` flag set, it may not be instantiated.
- If the definition has the `TD_ABSTRACT` flag set, it may not be instantiated.
- Otherwise, this is a class and may be instantiated.

If it turns out that the type may not be instantiated, the runtime shall throw the `RuntimeError` exception to indicate the problem.

The steps taken by the runtime to create the object, after validating that it is possible, are as follows:

1. Allocate a sub-object (hence including the data area) for the main class, and then for each of the interfaces it implements.
2. Repeat step 1 for each ancestor, starting from the parent, then the parent of the parent, etc, all the way until the parent is `NULL`. However, if an interface is implemented by one of the parents, and one of the children implements it and hence a sub-object for the interface was already created, *do not create it again*.
3. Zero out all data areas.
4. Create the `struct elec_object` and set `objRefCount` to 1, `objMainClass` to the class being instantiated, and enter the pointer to the array of sub-objects.
5. For each class in the hierarchy, starting at the *top* (from `elec.rt.Object`), call the *initializer function* specified in the class definition. The initializer expects the newly-created sub-object to be placed in the *this-register* before the call; you *do not* increase the reference count when calling the initializer, and the initializer does not decrease it. The initializer will initialize all fields with static initializers, and will also modify sub-objects belonging to its parents to override methods, and the sub-objects of interfaces to place implementations. The initializer may also call a constructor (in which case it may `incref` the object for the function call).
6. Once all initializers were called, you may return the object.

5.3. Destroying Objects

An object shall be destroyed when a `decref` results in the reference count becoming zero. When a binary notices that it has released the last reference to an object, it shall destroy it as follows:

1. Set the *this-register* to the pointer to the object description.
2. Call `_Elec_Destroy` in an architecture-specific way.

The runtime destroys the object by executing the following procedure:

1. For each sub-object starting from the main class, up to the topmost parent (i.e. in the order they are actually listed), call the *finalizer function* of the class. This is done by placing the pointer to the *data area* (not the object description being released!) in the *this-register* and then issuing an architecture-specific call to the finalizer. The finalizer shall decref all fields in the data area which contain managed objects.
2. All memory occupied by the object shall be freed.

5.4. Atomicity and Ownership Rules

The binary interface must be *interrupt-safe*, *thread-safe* and *reference-safe*. In this context, interrupt safety refers to the fact that it should be possible to throw an exception from an asynchronous signal (interrupt) handler. Thread safety refers to the fact that all operations on objects must be atomic, and must not be broken by interference from other threads. Reference safety refers to the fact that a reference count must not reach zero temporarily before being `incref`'ed again, as the `decref` would cause the object to be deleted. For this reason, the EGBI puts forward the following requirements:

- Once the object is created and is returned to the application, it is considered a shared resource and all accessed must be atomic.
- The list of sub-objects is immutable and does not need locks to be traversed.
- In the data area, methods (which are function pointers) are immutable once the object has been created, and can be read without locks.
- Fields in the data area must be accessed atomically as described below.
- Incrementing and decrementing the reference count must be done using atomic operations.
- Due to the atomicity of `decref`s, when the reference count turns out to be zero, this implies that no other thread is working on the object anymore, and so releasing it can occur without regard to concurrency.

5.5. Field Access

When assigning a managed object reference to a field, you transfer ownership of that reference to the field; so the object being in that field counts towards its reference count. You must atomically store the reference in the field, while reading out the old value, and if not `NULL`, decreasing its reference count. When reading a field, you must fetch and increment the reference count atomically, so that another thread setting its value to something else does not decrease the reference count before you increase it

(as this may cause the object to be deleted when you still need it). Exactly how those operations are implemented depends on the particular architecture.

5.5.1. Resolving a Field

In order to find out where a field is located, the compiler shall, at compile-time, find which class in the hierarchy of the object's type contains the named field. The emitted instructions should then:

1. Load the object whose field is to be accessed into the *this-register*.
2. Load the type definition of the class declaring the field into the *argument-register* (not as an array as would be done for a normal call).
3. Call `_Elec_GetData` in the architecture-specific way.
4. The *return-register* now contains a pointer to the data area belonging to the class in question. To this pointer, you add the byte offset indicated by the unatural_t value at the symbol named `_Elec_Field_#(`*`ClassName`*`)_`*`FieldName`*, where *ClassName* is the full name (path) of the class that declares the field (as previously decided by the compiler) and *FieldName* is the name of the field. The result is a pointer to the field.

5.5.2. Static Fields

A static field is simply located at the symbol `_Elec_Field_#(`*`ClassName`*`)_`*`FieldName`*.

5.6. Method Calls

Methods are defined by function pointers in a class or interface's data area, pointing to implementations of said methods. The location in the data area that stores the function pointer to the implementation of the method is known as the *method slot*. If a child class overrides a method, its initializer will find the method slot and replace it with its own implementation.

The procedure for finding the method slot is the same as finding a field, except the offset is stored instead by the symbol `_Elec_Method_#(`*`ClassOrInterfaceName`*`)_#(`*`MethodName`*`)_#(`*`Arg1TypeName`*`)_#(`*`Arg2TypeName`*`)_…`. For example, this method signature in class `hello.HelloWorld`:

```
void exampleMethod(Object a, String b)
```

Would have this offset symbol name:

```
_Elec_Method_5hello10HelloWorld_13exampleMethod_4elec2rt6Object_4elec2rt6String
```

Having found a method slot, you should first read its value and check if it is NULL; if so, the method is *unimplemented*, and the attempt to call it shall result in the `ImplementationError` exception being thrown.

5.6.1. Static Methods

A static method does not need a method slot. Instead, you simply call it by setting the *this-register* to NULL, passing arguments as per the calling convention (Section 3) and issuing an architecture-specific call to "`_Elec_Static_#(`*`ClassOrInterfaceName`*`)_#(`*`MethodName`*`)_#(`*`Arg1TypeName`*`)_#(`*`Arg2TypeName`*`)_…`".