Octopus DRM Technology Platform Specifications

Version 1.0.3
Final

Source	Marlin Developer Community
Date	February 3, 2010

Notice

Use of this document is subject to the agreement executed between you and the Parties, if any.

Any copyright notices shall not be removed, varied, or denigrated in any manner.

Intellectual Property

A commercial implementation of this specification requires a license from the Marlin Trust Management Organization.

Contact Information

Feedback on this specification should be addressed to:
editor@marlin-community.com

Contact information for t stop he Marlin Trust Management Organization can be found at: http://www.marlin-trust.com/

Table of Contents

1 Introduction

This document contains the Octopus DRM technology platform specifications.

Please note: This document consolidates material that was previously in separate documents. The highest version number among the source documents consolidated into this document was 1.0.2. The version number for this document is initially one greater than that in the third digit, that is, 1.0.3.

Octopus is a general-purpose DRM architecture that can be applied to a variety of applications ranging from enterprise document control to medical record privacy protection, consumer media copyright protection, or any other system requiring distributed governance and control of information. At the center of an Octopus system is an Octopus DRM Engine—a small, lightweight component responsible for determining whether access to content should be granted in a given set of conditions. By itself, Octopus is entirely agnostic regarding content formats, hardware/software platforms, and business semantics. That is, the Octopus DRM Engine responsible for governing access to content is entirely unaware of the type of content it protects, and is not bound by a particular set of semantics (e.g., not limited by predefined meanings or hard-coded “rules languages”).

One of the distinguishing features of an Octopus-based system is its ability to separate the *protection* of content from the *governance* of that content. This allows, among other benefits, the ability to issue rights to access content separately from information that governs where or when it can be used. This enables great end-user flexibility; for example, a service provider can issue rights to a customer to use content, but can allow that customer to independently manage which devices he or she would like to use that content on at any particular moment.

There are six Octopus specifications, all of which are defined in this document:

1. Octopus Objects (see §2)

   This specification describes the basic objects that are the building blocks of Octopus. It first shows the high-level view of which types of objects Octopus uses for content protection and governance, and how they relate to one another, followed by a more detailed description of those objects and the information they convey. It then describes the objects used for Rules Conditions, Identity and Key Management.

2. Octopus Controls (§3)

   This specification describes the Control objects in detail. Control objects can be used to represent rules that govern access to content by granting or denying the use of the ContentKey objects they control. They can also be used to limit the validity of a Link object in which they are embedded. This specification defines which actions the application can perform on the content, which action parameters should be supplied to the control program, and how the control program encodes the return status indicating that the requested action can or cannot be performed.

3. Plankton Virtual Machine (§4)

   This specification defines Plankton, the virtual machine (VM) used by the Octopus Engine to execute control programs that govern access to content. It is used by the DRM Client Engine to execute control programs.

4. Octopus Object Serialization (§5)

This specification defines an encoding-neutral way of computing a canonical byte sequence for Octopus objects. The purpose of this canonical byte sequence is the computation of a digest for the digital signature of objects.

5. Scuba Key Distribution (§6)

   This specification defines Scuba, a key distribution system that has been designed to fit very naturally within the Octopus architecture. The basic principle behind Scuba is to use the Octopus Link objects to distribute keys, in addition to their primary purpose of establishing relationships between Node objects. An Octopus Control object contains a control program that decides whether or not a requested action should be granted. That control program often checks that a specific Octopus Node is reachable via a collection of Octopus Links. Scuba makes it possible to take advantage of the existence of that collection of Links to facilitate the distribution of a key such that it is available to the Octopus Engine that is executing the control.

6. SeaShell Object Store (§7)

   This specification defines a secure Object Store that can be used by Octopus Engine implementations to provide a secure state storage mechanism. Such a facility is useful to enable control programs to read and write in a protected state database that is persistent from invocation to invocation.

The final section of this document, §8, provides a table with complete references for the external documents referred to within this document.

1.1 *Definitions, Acronyms and Abbreviations*

byte	8-bit value, or octet
byte code	Stream of bytes that encode executable instructions and their operands
DRM	Digital Rights Management
ESB	Extended Status Block, as defined in §4.7.3.2
LSB	Least Significant Bit
EC	Program Counter
SP	Stack Pointer
UTC	Coordinated Universal Time
VM	Virtual Machine

1.2 *Conformance Conventions*

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this specification are to be interpreted as described in IETF RFC 2119 [RFC2119].

2 Octopus Objects

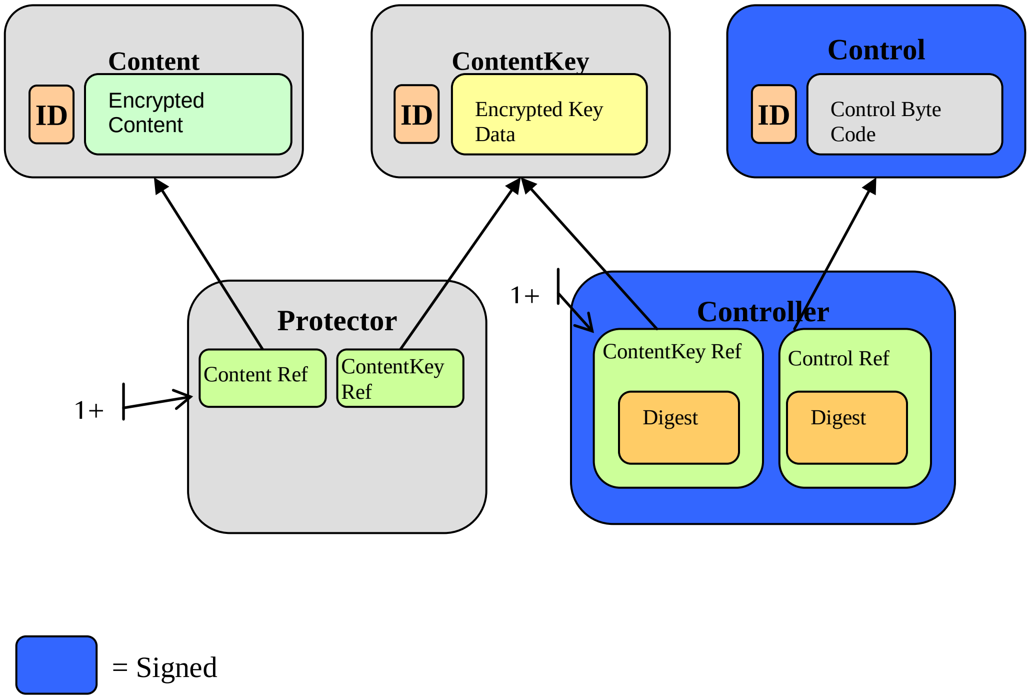
2.1 *Introduction*

This section contains the Octopus Objects specification, which describes the basic objects that are the building blocks of Octopus. We first show the high-level view of which types of objects Octopus uses for content protection and governance, and how they relate to one another, followed by a more detailed description of those objects and the information they convey. We then describe the objects used for Rules Conditions, Identity and Key Management.

This specification defines the object model, but not the encoding of the objects. The data structures defined in §2.4 are designed to be encoded or serialized according to one or more schemas that may be defined in other specifications. Likewise, the algorithms and encodings used for signing and encrypting object data are also not defined by this specification.

2.2 *Content Protection and Governance Objects*

The content governance objects are the objects that are used to protect content and associate usage rules (controls) to protected content. Together, those objects form what is referred to as a ‘license’.



The data represented by the Content object are encrypted by a key. That key is represented by a ContentKey object, and the binding between the content and the key used to encrypt it is represented by the Protector object. The rules that govern the use of the key to decrypt the content are represented by the Control object, and the binding between the ContentKey and the Control used to govern its use is represented by the Controller object. All compliant systems

SHALL only make use of the content decryption key under governance of the rules expressed by the byte code in the Control object.

2.2.1 Common Elements

All Octopus objects share common basic traits: they can each have an ID, a list of attributes, and a list of extensions.

2.2.1.1 IDs

All objects that are referenced by other objects have a unique ID. IDs are simply URIs, and the convention in Octopus is that those URIs are URNs.

2.2.1.2 Attributes

Attributes are typed values. Attributes can be named or unnamed. The name of a named attribute is a simple string or URI. The value of an attribute is of a simple type (string, integer and bytes) or a compound type (list and array). Attributes of type ‘list’ contain an unordered list of named attributes. Attributes of type ‘array’ contain an ordered array of unnamed attributes.

An object’s ‘attributes’ field is a (possibly empty) unordered collection of named attributes.

2.2.1.3 Extensions

Extensions are elements that can be added to objects to carry optional or mandatory extra data. Extensions are typed, and have unique IDs. Extensions can be internal or external.

2.2.1.3.1 Internal Extensions

Internal extensions are contained in the object they extend. They have a ‘critical’ flag that indicates whether the specific extension data type for the extension is required to be known to the implementation that uses the object. If an implementation encounters an object with a critical extension with a data type that it does not understand, it MUST reject the entire object.

The ID of an internal extension MUST be locally unique: an object cannot contain two extensions with the same ID, but it is possible that two different objects each contain an extension with the same ID as that of an extension of the other object.

An object’s ‘extensions’ field is a (possibly empty) unordered collection of internal extensions.

2.2.1.3.2 External Extensions

External extensions are not contained in the object they extend. They appear independently of the object, and have a ‘subject’ field that contains the ID of the object they extend.

The ID of an external extension MUST be globally unique.

2.2.2 Content

The Content object is an “external” object. Its format and storage are not under the control of Octopus, but rather under the control of the content management subsystem of the host application. (For instance, it could be an MP4 movie file, an MP3 music track, etc.) The format for the content needs to provide support for associating an ID with the content payload data. The content payload is encrypted in a format-dependent manner (typically with a symmetric cipher, such as AES).

2.2.3 ContentKey

The ContentKey object represents a unique encryption key, and associates an ID with it. The purpose of the ID is to enable Protector objects and Controller objects to make references to ContentKey objects. The actual key data encapsulated in the ContentKey object is itself encrypted so that it can only be read by the recipients that are authorized to decrypt the content. The ContentKey object specifies which cryptosystem was used to encrypt the key data. The cryptosystem used to protect the content key data is called the Key Distribution System. Different Key Distribution Systems can be used. An example of a Key Distribution System is the Scuba Key Distribution System, described in §6.

2.2.4 Protector

The Protector object contains the information that makes it possible to find out which key was used to encrypt the data of Content objects. It also contains information about which encryption algorithm was used to encrypt that data. The Protector object contains one or more IDs that are references to Content objects, and exactly one ID that is a reference to the ContentKey object that represents the key that was used to encrypt the data. If the Protector points to more than one Content object, all those Content objects represent data that has been encrypted using the same encryption algorithm and the same key. Unless the cryptosystem used allows a safe way of using the same key for different data items, it is NOT RECOMMENDED that a Protector object point to more than one Content object.

2.2.5 Control

The Control object contains the information that allows Octopus to make decisions regarding whether certain actions on content should be permitted when requested by the host application. The rules that govern the use of content keys are encoded in the Control object as Plankton byte code. (See §4 for the Plankton Virtual Machine specification.) The Control object also has a unique ID so that it can be referenced by a Controller object. Control objects MUST be signed, so that Octopus can verify that the control byte code is valid and trusted before it is used to make any decisions. A signature of a Control is either direct or indirect. A direct signature is a signature of a Control object itself. An indirect signature exists when a signed Controller object has a reference to a Control. (In this case, the ControlRef field of the Controller object contains a digest of the Control object.)

2.2.6 Controller

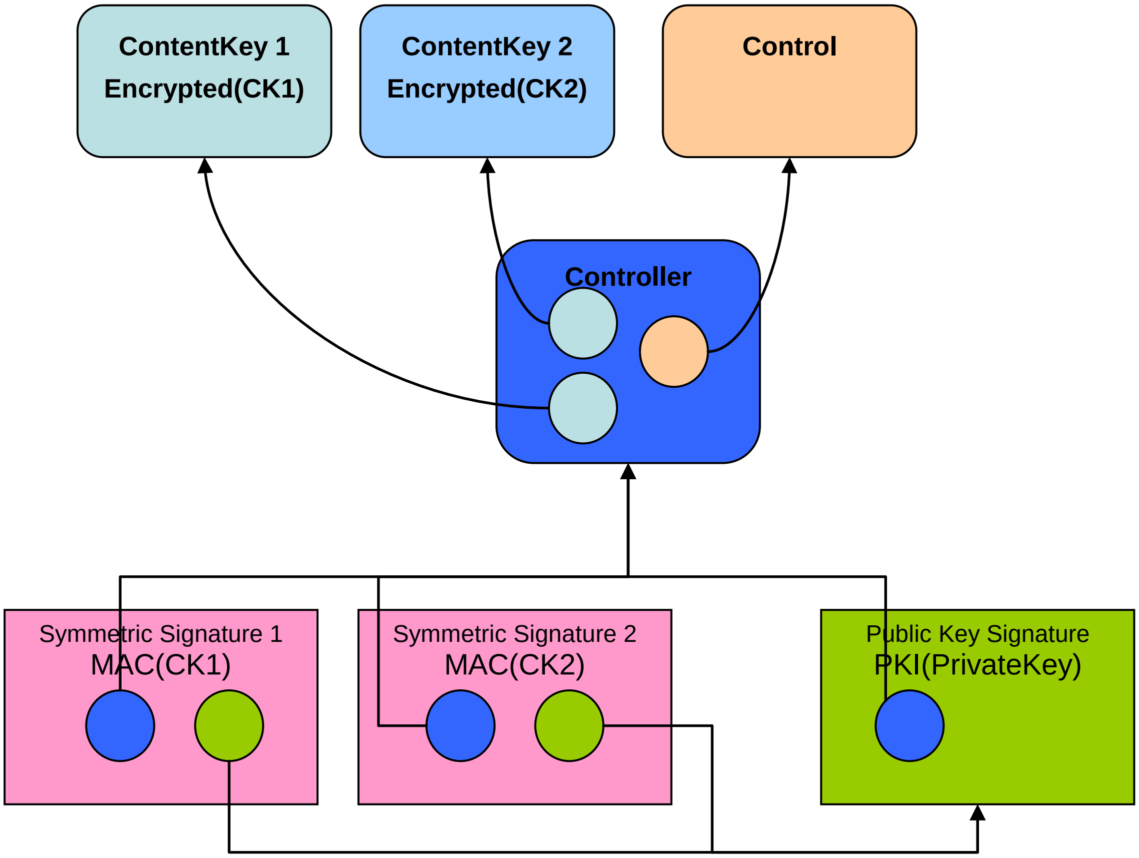
The Controller object contains the information that allows Octopus to find out which Control governs the use of one or more keys represented by ContentKey objects. It contains information that binds it to the ContentKey objects and the Control object that it references. Controller objects MUST be signed, so that the validity of the binding between the ContentKey and the Control object that governs it, as well as the validity of the binding between the ContentKey ID and the actual key data, can be established. A signature of the Controller object can be a public key signature or a symmetric key signature, or a combination of both. Also, when the digest of the Control object referenced by the Controller object is included in the Controller object, the validity of the Control object can be derived without having to separately verify the signature of the Control object.

2.2.6.1 Symmetric Key Signature

This is the most common type of signature for Controller objects. This type of signature is implemented by computing a MAC (Message Authentication Code) of the Controller object, keyed with the content key, that is, the same key as the key represented by the ContentKey object. The canonical method for this MAC is to use HMAC with the same hashing algorithm(s) as the one(s) chosen for the crypto algorithms used in the same Octopus deployment. There MUST be one symmetric key signature of a Controller object for each ContentKey object referenced by that Controller.

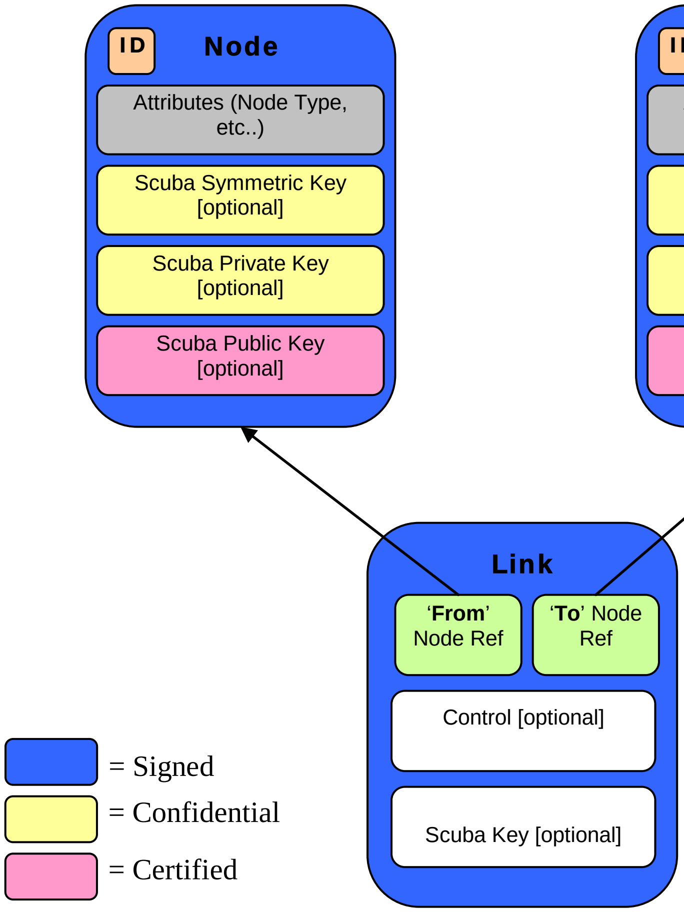
2.2.6.2 Public Key Signature

This type of signature is used when the identity of the signer of the Controller object needs to be known. This type of signature is implemented with a public key signature algorithm, signing with the private key of the principal who is asserting the validity of this object. When using this type of signature, a symmetric key signature SHALL also be present, and it SHALL sign both the Controller object and the public key signature, so that it can be guaranteed that the principal who signed with its private key also had knowledge of the actual value of the content key carried in the ContentKey object.



Example of multiple signatures for a Controller object

2.3 *Rule Conditions, Identity and Key Management Objects*


ID
Node
Attributes (Node Type, etc..)
Scuba Symmetric Key [optional]
Scuba Private Key [optional]
Scuba Public Key [optional]
ID
Node
Attributes (Node Type, etc..)
Scuba Symmetric Key [optional]
Scuba Private Key [optional]
Scuba Public Key [optional]
Link
'From' Node Ref
'To' Node Ref
Control [optional]
Scuba Key [optional]
= Signed
= Confidential
= Certified

Node objects represent entities in an Octopus deployment (a DRM system that uses Octopus). Octopus does not have implicit or explicit semantics for what the Node objects represent. A given Octopus deployment will define what types of principals exist, and what roles and identities different Node objects represent. That semantic information is typically expressed using Attributes of the Node object. **Link** objects represent relationships between Nodes. Link objects can also optionally contain some cryptographic data that allow Octopus to use the Links for ContentKey derivation computations (see §6). Just as for Nodes, Octopus does not have implicit or explicit semantics for what a Link relationship means. Depending on what the *from* and *to* Nodes of the Link represent in a given deployment, the meaning of the Link relationship can express membership, ownership, association, and many other types of relationships. In a typical Octopus deployment, some Node objects could represent Users, other Nodes would represent Devices, and other Nodes would represent User Groups or Authorized Domains (ADs). In that context, Links between Devices and Users would represent an ownership relationship, and Links between Users and User Groups or ADs would represent membership relationships.

2.3.1 Node

A Node object represents an entity in the system. A Node object's Attributes define certain aspects of what the Node object represents, such as the role or identity represented by the Node object in the context of a specific deployment.

2.3.2 Link

A Link object is a signed assertion that there exists a directed edge in the graph whose vertices are the Node objects. For a given set of Nodes and Links, we say that there is a *path* between a Node X and a Node Y if there exists a directed path between the Node X vertex and the Node Y vertex in the graph. When there is a path between Node X and Node Y, we say that Node Y is *reachable* from Node X. Those assertions represented by Link objects are used to express which Nodes are reachable from other Nodes. The controls that govern Content objects can check, before they allow an action to be performed, that certain Nodes are reachable from the Node associated with the entity performing the action. For example, if Node D represents a device that wants to perform the 'Play' action on a Content object, a control that governs this Content object can test if a certain Node U representing a certain user is reachable from Node D. To determine if Node U is reachable, Octopus will check if there exists a set of Link objects that can establish a path between Node D and Node U.

All Links MUST be signed. Octopus verifies Link objects before it can use them to decide the existence of paths in the Node graph. Depending on the specific features of the certificate system (for example, X.509 v3) used to sign Link objects, Link objects can be given limited lifetimes, be revoked, etc. Also, the policies that govern which keys can sign Link objects, which Link objects can be created, and the lifetime of Link objects are not defined by this specification. Those policies will exist outside of the scope of this specification, and will typically leverage the Node Attributes information.

A Link object MAY contain a Control object that will be used to constrain the validity of the Link, as specified in the Octopus Controls specification in §3.

2.4 *Data Structures*

The remaining sections of this specification define in more detail the object model for the Octopus objects, defining the fields of each type of object.

These data structures are described using a simple object description syntax. Each object type is defined by a class that can extend a parent class (this is an 'is-a' relationship). The class descriptions are in terms of the simple abstract types 'string' (character string), 'int' (integer value), 'byte' (8-bit value), and 'boolean' (true or false) but this specification does not define any specific encoding for those data types, or for compound structures containing those types. The way objects are encoded, or represented, can vary depending on the implementation of the Engine. Typically, a given Octopus deployment will specify how the fields are represented (for example, using an XML schema).

The following notations are used:

`class ClassName {`
`    field1;`
`    field2;`
`    ...`
`}`
	Defines a class type. A class type is a heterogeneous compound data type (also called an object type). This compound type is made up of one or more fields, each of a simple or compound type. Each field can be of a different type.

`type[]`	Defines a homogeneous compound data type (also called a list or array type). This compound type is made up of 0 or more elements of the same type (0 when the list is empty).
`string`	Simple type: represents a Unicode character string
`int`	Simple type: represents an integer value between -2147483648 and 2147483647
`byte`	Simple type: represents an integer value between 0 and 255
`boolean`	Simple type: represents a Boolean value (true or false)
`class SubClass extends SuperClass {…}`	Defines a class type that extends another class type. A class that extends another one contains all the fields of the class it extends (called the superclass) in addition to its own fields.
`abstract class {…}`	Defines an abstract class type. Abstract class types are types that can be extended, but are never used by themselves.
`{type field;}`	Defines an optional field. An optional field is a field that may be omitted from the compound data type that contains it.
`(type field;)`	Defines a field that will be skipped when computing the canonical byte sequence for the enclosing compound field (see §5).
`class SubClass extends SuperClass(field=value) {…}`	Defines a subclass of a class type and specifies that for all instances of that subclass, the value of a certain field of the superclass is always equal to a fixed value.

2.4.1 Common Structures

```
abstract class Octobject {
    {string id;}
    Attribute[] attributes;
    InternalExtension[] extensions;
}

class Transform {
    string algorithm;
}

class Digest {
    Transform[] transforms;
    string algorithm;
    byte[] value;
}

class Reference {
    string id;
    {Digest digest;}
}
```

2.4.1.1 Attributes

Attributes are either named or unnamed. Named attributes have a non-empty name field. Unnamed attributes do not have a name. All attributes have a type. There are five attribute types: IntegerAttribute, StringAttribute, ByteArrayAttribute, ListAttribute and ArrayAttribute. Attributes of type IntegerAttribute, StringAttribute and ByteArrayAttribute are value-type attributes: they have a value field. Attributes of type ListAttribute and ArrayAttribute are container-type attributes: they contain other attributes. Attributes of type ListAttribute MUST only contain named attributes. Attributes of type ArrayAttribute MUST only contain unnamed attributes.

```
abstract class Attribute {
    {string name;}
    string type;
}

class IntegerAttribute extends Attribute(type=’int’) {
    int value;
}

class StringAttribute extends Attribute(type=’string’) {
    string value;
}

class ByteArrayAttribute extends Attribute(type=’bytes’) {
    byte[] value;
}

class ListAttribute extends Attribute(type=’list’) {
    Attribute[] attributes; // must all be named
}

class ArrayAttribute extends Attribute(type=’array’) {
    Attribute[] attributes; // must all be unnamed
}
```

2.4.1.2 Extensions

There are two types of extensions:

Internal Extensions: carried inside the Octobject

External Extensions: carried outside the Octobject

```
abstract class ExtensionData {
    string type;
}

abstract class Extension {
    string id;
}

class ExternalExtension extends Extension {
    string subject;
    ExtensionData data;
}

class InternalExtension extends Extension {
    boolean critical;
    {Digest dataDigest;}
    (ExtensionData data;)
}
```

It is important to be able to verify the signature of an Octobject even if a particular type of ExtensionData is not understood by a given implementation. This is the purpose of the level of indirection added by the ‘dataDigest’ field.

If the specification of this ExtensionData mandates that the data are part of the signature within the context of a particular Octobject, then the ‘dataDigest’ field MUST be present. An implementation that understands this ExtensionData, and is therefore capable of computing its canonical representation, can then verify the digest.

If the specification of this ExtensionData mandates that the data are NOT part of the signature, then the ‘dataDigest’ field MUST NOT be present.

2.4.2 Node Objects

```
class Node extends Octobject {
}
```

2.4.3 Link Objects

```
class Link extends Octobject {
    string fromId;
    string toId;
    {Control control;}
}
```

2.4.4 Control Objects

```
class Control extends Octobject {
    string protocol;
    string type;
    byte[] codeModule;
}
```

2.4.5 ContentKey Objects

```
abstract class Key {
    string id;
    string usage;
    string format;
    byte[] data;
}

abstract class PairedKey extends Key {
    string pairId;
}

class ContentKey extends Octobject {
    Key secretKey;
}
```

Each key has a unique id, a format, a usage (which MAY be an empty string), and data. The ‘usage’ field, if non-empty, specifies the purpose for which this key can be used. For content keys, the ‘usage’ field MUST be empty. For Scuba (§6), this field specifies whether this is a key sharing key or a confidentiality key. The ‘format’ field specifies the format of the ‘data’ field (such as, for example, ‘RAW’ for symmetric keys, or ‘PKCS#8’ for RSA private keys, etc.). The ‘data’ field contains the actual key data, formatted according to the format specified in the ‘format’ field.

For keys that are part of a key pair (such as RSA keys), the extra field ‘pairId’ gives a unique identifier for the pair, so that the pair can be referenced from other data structures.

NOTE: The ‘data’ field in the Key object is the plaintext value of the actual key, even if the object’s representation contains an encrypted copy of the key.

2.4.6 Controller Objects

```
class Controller extends Octobject {
    Reference controlRef;
    Reference[] contentKeyRefs;
}
```

The ‘digest’ field of the ‘controlRef’ MUST be present.

2.4.7 Protector Objects

```
class Protector extends Octobject {
    Reference contentKeyRef;
    Reference[] contentRefs;
}
```

The ‘digest’ field of the ‘contentKeyRef’ field and of all the elements of the ‘contentRefs’ field MUST be omitted.

3 Octopus Controls

3.1 *Introduction*

This section of the document provides the specification for Octopus Control objects, describing them in detail. Control objects can be used to represent rules that govern access to content by granting or denying the use of the ContentKey objects they control. They can also be used to represent constraints on the validity of a Link object in which they are embedded, or as standalone program containers that are run on behalf of another entity, such as in agents or delegates. Controls contain metadata and byte-code programs, which implement a specific interaction protocol. The purpose of a Control Protocol is to specify the interaction between the Octopus Engine and a control program or between a host application and a control program through the Octopus Engine. This specification also defines which actions the application can perform on the content, which action parameters should be supplied to the control program, and how the control program encodes the return status indicating that the requested action can or cannot be performed, as well as parameters that can further describe the return status.

3.2 *Control Programs*

A Control object contains a control program. A control program contains a Plankton code module. (Plankton is described in §4.) A code module contains executable byte-code, and a list of named routines (in the Export Table entries).

3.2.1 Naming Conventions

The set of routines that represent the rules that govern the performance of a certain operation (such as “play”) on a content item is called an ‘action control’. The set of routines that represent a validity constraint on a Link object is called a ‘Link constraint’. The set of routines that are intended to be executed on behalf of a remote entity (such as during a protocol session with an Octopus Engine running on a different host) is called an ‘agent’. The set of routines that are intended to be executed on behalf of another control (such as when a control program uses the System.Host.CallVm system call) is called a ‘delegate’.

3.2.2 Interface to Control Programs

Control programs are executed by a Plankton VM running in a Plankton host environment. The specification does not make specific assumptions regarding the implementation of the host environment other than the fact that it complies with the Plankton specification. However, for the purpose of clarity, it is assumed that the implementation of the VM’s host environment can be logically separated into two parts: a host application, and a DRM engine that we call the Octopus Engine. An implementation MAY have a different logical separation of functions, provided that the resulting behavior of the system is functionally equivalent to what is described in this specification.

The Octopus Engine is the logical interface between the host application and the control programs. The host application makes logical requests to the engine, such as requesting access to a content key for a certain purpose (ex: to play or render a content stream). The engine MUST ensure that the interaction protocol defined in this specification is implemented correctly, such as ensuring any guarantees regarding a control program’s initialization, call sequence, and other interaction details.

When the host application requests the use of content keys for a set of content IDs, the Octopus Engine determines which Control object must be used. The Protector object allows the engine to resolve which ContentKey objects need to be accessed for the requested content IDs. The engine then finds the Controller object that references those ContentKey objects. The object model states that a Controller object can reference more than one ContentKey object. This allows multiple ContentKey objects to be governed by the same Control object. When the host application requests access to a content key by invoking an action, it can request multiple content IDs as a group, to the extent that the ContentKey objects that correspond to them are all referenced by the same Controller object. It is not possible to request access to a group of content keys referenced by more than one Controller object.

The Octopus Engine follows a convention for mapping actions to routine names. For each of the routines described below, the name that appears in the Export Table entry in the code module is the string equal to the routine name as it is spelled in the section titles.

3.2.2.1 Control Loading

Before the engine can make calls to control routines, it must load the control's code module into a Plankton VM. In this specification, only one code module per VM is loaded. Therefore, all control routines SHALL be running in a VM with no other code module loaded.

3.2.2.2 Atomicity

The engine MUST ensure that calls to control routines are atomic with respect to the resources it makes available to the routines, such as the SeaShell databases. (SeaShell is described in §7.) This means that the engine MUST ensure that those resources remain unmodified during the execution of any of the routines it calls. This may be done by effectively locking those resources during a routine call, or by preventing multiple VMs from running concurrently. However, the engine need not guarantee that those resources are unmodified across successive routine invocations.

3.2.3 Control Protocol

The routine naming, and the input/output interface and data structures for each routine in a code module, together constitute a Control Protocol. The protocol implemented by a code module is signaled in the Control object's 'protocol' field. The Control Protocol described in this document is called the Standard Control Protocol, and its identifier (the value of the 'protocol' field) is 'http://www.octopus-drm.com/specs/scp-1_0'.

Before the Octopus Engine loads a code module and calls routines in the control program, it needs to guarantee that the interaction with the control program will be consistent with the specification for the specific protocol ID signaled in the 'protocol' field. That includes any guarantee about the features of the Plankton Virtual Machine that need to be implemented, guarantees about the size of the address space available to the control program, etc.

It is possible for control protocols, such as the Standard Control Protocol, to evolve over time without having to create a new protocol specification. As long as the changes made to the protocol are consistent with previous revisions of the specification and it can be guaranteed that existing implementations of the Octopus Engine, as well as existing control programs that comply with that protocol, will continue to perform according to the specification, then the changes are deemed compatible. Such changes MAY include, for instance, new action types.

3.2.4 Byte Code Type

The type of the byte-code module used in this protocol is Plankton byte-code module version 1.0 as specified in §4. The value for the ‘type’ field of the Control object is ‘http://www.octopus-drm.com/specs/pkcm-1_0’.

3.2.5 General Control Routines

General routines are routines that are applicable to the control as a whole, and are not specific to a given action or Link constraint.

3.2.5.1 Control.Init

This routine is OPTIONAL. If this routine exists, the engine MUST call it exactly once before any other control routine is called.

Input: None

Output:

Top of stack:

ResultCode
…

ResultCode: 0 on success, or a negative error code on failure.

If ResultCode is not 0, the engine MUST abort the current control operation and MUST NOT make any further calls to routines for this control.

3.2.5.2 Control.Describe

This routine is OPTIONAL. The routine is called when the application requests a description of the meaning of the rules represented by this control program in general (i.e., not for a specific action).

Input: None

Output:

Top of stack:

ResultCode
StatusBlockPointer
…

ResultCode: An integer value. The result value is 0 if the routine completed successfully, or a negative error code if it did not.

StatusBlockPointer: Address of a standard ExtendedStatusBlock (see §4.7.3.2).

3.2.5.3 Control.Release

This routine is OPTIONAL. If this routine exists, the engine MUST call it exactly once after it no longer needs to call any other routine for this control. No other routine will be called for this control unless a new use of the control is initiated (in which case, the Init routine MUST be called again).

Input: None

Output:

Top of stack:

ResultCode
…

<u>*ResultCode*</u>: 0 on success, or a negative error code on failure.

If ResultCode is not 0, the engine MUST NOT make any further calls to routines for this control.

3.2.6 Action Routines

Each possible action has a name. For a given action <Action>, the following routine names are defined (<Action> stands for the actual name of the action):

3.2.6.1 Control.Actions.<Action>.Init

This routine is OPTIONAL. If this routine exists, the engine MUST call it exactly once before any other routine is called for this action.

Input: None

Output:

Top of stack:

ResultCode
…

<u>*ResultCode*</u>: 0 on success, or a negative error code on failure.

If ResultCode is not 0, the engine MUST abort the current action operation and MUST NOT make any further calls to routines for this action.

3.2.6.2 Control.Actions.<Action>.Check

This routine is REQUIRED. This routine is called to check what the return status would be if the Perform routine were to be called, without actually performing the action. This routine MUST NOT have any side effects.

NOTE: If the Perform routine has no side effects, the Check and Perform entries in the control's Export Table can point to the same routine.

This routine has the same inputs and outputs as the Perform routine described in §3.2.6.3.

3.2.6.3 Control.Actions.<Action>.Perform

This routine is REQUIRED. This routine is called when the application is about to perform the action.

NOTE: When an action is performed, only the ‘Perform’ routine is called. The engine SHALL NOT call the ‘Check’ routine before calling the ‘Perform’ routine. The implementation of the ‘Perform’ can call the ‘Check’ routine internally if it chooses to, but should not assume that the system will have called it beforehand.

Input: None

Output:

Top of stack:

ResultCode
StatusBlockPointer
…

<u>ResultCode</u>: An integer value. The result value is 0 if the routine was able to run, or a negative error code if an error occurred.

A success ResultCode (0) return does not mean that the action request is granted. It only means that the routine was able to run without error. It is the StatusBlock that indicates whether the request is granted or denied.

<u>StatusBlockPointer</u>: Address of a standard ExtendedStatusBlock (see §4.7.3.2).

Description: A host application MUST call this routine when it is about to perform an action and use the value of a ContentKey object. If the ResultCode indicates a failure, or the StatusBlock’s category is ACTION_DENIED, or the returned ESB must be rejected (as described in §3.3.4.3), the host application is not authorized to use the value of the ContentKey for the requested action and MUST abort the action.

3.2.6.4 Control.Actions.<Action>.Describe

This routine is OPTIONAL. The routine is called when the application requests a description of the meaning of the rule represented by this control program for this action.

Input: None

Output:

Top of stack:

ResultCode
StatusBlockPointer
…

<u>ResultCode</u>: An integer value. The result value is 0 if the routine was able to run, or a negative error code if an error occurred.

<u>StatusBlockPointer</u>: Address of a standard ExtendedStatusBlock (see §4.7.3.2).

3.2.6.5 Control.Actions.<Action>.Release

This routine is OPTIONAL. If this routine exists, it is called exactly once after the Octopus Engine no longer needs to call any other routine for this action. No other routine will be called for this action unless a new use of the action is initiated (in which case, the action's Init routine will be called again).

Input: None

Output:

Top of stack:

ResultCode
…

<u>*ResultCode*</u>: 0 on success, or a negative error code on failure.

If ResultCode is not 0, the engine MUST NOT make any further calls to routines for this action.

3.2.7 Link Constraint Routines

When a Link object has an embedded Control, the Octopus Engine MUST call the Link Constraint Check routine in that control (§3.2.7.2) to verify the validity of that Link object.

3.2.7.1 Control.Link.Constraint.Init

This routine is OPTIONAL. If this routine exists, it is called exactly once before any other routine is called for this Link constraint.

Input: None

Output:

Top of stack:

ResultCode
…

<u>*ResultCode*</u>: 0 on success, or a negative error code on failure.

If ResultCode is not 0, the engine MUST consider that the validity constraint for the Link object under evaluation is not satisfied, and MUST NOT make any further calls to routines for this Link control.

3.2.7.2 Control.Link.Constraint.Check

This routine is REQUIRED. This routine is called to check if the validity constraint for this Link is satisfied.

Input: None

Output:

Top of stack:

ResultCode
StatusBlockPointer
…

<u>ResultCode</u>: An integer value. The result value is 0 if the routine was able to run, or a negative error code if an error occurred.

NOTE: A success ResultCode (0) return does not mean that the constraint is satisfied. It only means that the routine was able to run without error. It is the StatusBlock that indicates whether the constraint is satisfied or not.

<u>StatusBlockPointer</u>: Address of a Standard ExtendedStatusBlock (see §4.7.3.2).

If ResultCode is not 0, the engine MUST consider that the validity constraint for the Link object under evaluation is not satisfied, and MUST NOT make any further calls to routines for this Link control.

3.2.7.3 Control.Link.Constraint.Describe

This routine is OPTIONAL. The engine calls this routine when the application requests a description of the meaning of the constraint represented by this control program for this Link.

Input: None

Output:

Top of stack:

ResultCode
StatusBlockPointer
…

<u>ResultCode</u>: An integer value. The result value is 0 if the routine was able to run, or a negative error code if an error occurred.

<u>StatusBlockPointer</u>: Address of a Standard ExtendedStatusBlock (see §4.7.3.2).

3.2.7.4 Control.Link.Constraint.Release

This routine is OPTIONAL. If this routine exists, the engine MUST call it exactly once after it no longer needs to call any other routine for this constraint. No other routine can be called for this constraint unless a new cycle is initiated (in which case, the constraint’s Init routine MUST be called again).

Input: None

Output:

Top of stack:

ResultCode
…

__ResultCode__: 0 on success, or a negative error code on failure.

If ResultCode is not 0, the engine MUST NOT make any further calls to routines for this Link constraint.

3.2.8 Agent Routines

An agent is a Control object that is designed to run on behalf of an entity. Agents are typically used in the context of a service interaction between two endpoints, where one endpoint needs to execute some Plankton code within the context of the second endpoint, and possibly obtain the result of that execution. A control can contain multiple agents, and each agent can contain any number of routines that can be executed, but in practice, agents are likely to have a single routine.

The following entry points are defined for agents. <Agent> is a name string that refers to the actual name of an agent.

3.2.8.1 Control.Agents.<Agent>.Init

This routine is OPTIONAL. If this routine exists, the engine MUST call it exactly once before any other routine is called for this agent.

Input: None

Output:

Top of stack:

ResultCode
…

__ResultCode__: 0 on success, or a negative error code on failure.

3.2.8.2 Control.Agents.<Agent>.Run

This routine is REQUIRED. This routine is the main routine of the agent.

Input: None

Output:

Top of stack:

ResultCode
ReturnBlockAddress
ReturnBlockSize
…

__ResultCode__: An integer value. The result value is 0 if the routine was able to run, or a negative error code if an error occurred.

__ReturnBlockAddress__: Address of a block of memory that contains data that the agent code is expected to return to the caller. If the routine does not need to return anything, the address is 0.

<u>*ReturnBlockSize*</u>: Size in bytes of the block of memory at ReturnBlockAddress. If ReturnBlockAddress is 0, this value MUST also be 0.

3.2.8.3 Control.Agents.<Agent>.Describe

This routine is OPTIONAL. The routine is called when the application requests a description of the agent.

Input: None

Output:

Top of stack:

ResultCode
StatusBlockPointer
…

<u>*ResultCode*</u>: An integer value. The result value is 0 if the routine completed successfully, or a negative error code.

<u>*StatusBlockPointer*</u>: Address of a standard ExtendedStatusBlock (see §4.7.3.2).

3.2.8.4 Control.Agents.<Agent>.Release

This routine is OPTIONAL. If this routine exists, the engine MUST call it exactly once after it no longer needs to call any other routine for this agent. No other routine will be called for this agent unless a new cycle is initiated (in which case, the agent's Init routine MUST be called again).

Input: None

Output:

Top of stack:

ResultCode
…

<u>*ResultCode*</u>: 0 on success, or a negative error code on failure.

3.3 *Extended Status Blocks*

The following definitions are applicable to the ESB data structures returned by several of the routines described above. The ESB data structure is defined in the Plankton specification in §4.7.3.2.

3.3.1 Global Flags

There are no global ESB flags defined in this specification. Therefore all control programs compliant with this specification SHALL always set the GlobalFlags field of ESBs to 0.

3.3.2 Categories

The following sections 3.3.2.1 through 3.3.2.3 define values for the Category field of Extended Status Blocks. None of those categories has a subcategory; therefore the value of the SubCategory field of the ESBs SHALL be set to 0.

3.3.2.1 Check and Perform Routines for Actions

<table>
<tr><th>Value</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>ACTION_GRANTED</td><td>The application is authorized to use the content keys controlled by this control program for the purpose of the requested action.
The parameter list of the returned ESB SHOULD NOT contain any of the constraint parameters, but MAY contain obligation and/or callback parameters.
Subcategories:
<table>
<tr><th>Id</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>UNSPECIFIED</td><td>Unspecified (N/A)</td></tr>
</table></td></tr>
<tr><td>1</td><td>ACTION_DENIED</td><td>The application is not authorized to use the content keys controlled by this control program for the purpose of the requested action.
When an action is denied, the control program SHOULD include in the parameter list of the returned ESB one or more of the constraints that were not met and caused the action to be denied. (The constraints that were not evaluated and the constraints that did not cause the action to fail SHOULD be omitted.)
The parameter list of the returned ESB MUST NOT contain any obligation or callback parameter.
Subcategories:
<table>
<tr><th>Id</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>UNSPECIFIED</td><td>Unspecified (N/A)</td></tr>
</table></td></tr>
</table>

In the context of ESB parameters returned by action routines, a constraint means a condition that is required to be true or a criterion that is required to be met in order for the result of the routine to return an ESB with the Category ACTION_GRANTED.

Values for the LocalFlags field common to both categories described above:

Bit Index (0 is LSB)	Name	Description
0	OBLIGATION_NOTICE	The parameter list contains one or more parameters that are related to obligations.

1	CALLBACK_NOTICE	The parameter list contains one or more parameters that are related to callbacks.
2	GENERIC_CONSTRAINT	The parameter list contains one or more parameters that are related to generic constraints.
3	TEMPORAL_CONSTRAINT	The parameter list contains one or more parameters that are related to temporal constraints.
4	SPATIAL_CONSTRAINT	The parameter list contains one or more parameters that are related to spatial constraints.
5	GROUP_CONSTRAINT	The parameter list contains one or more parameters that are related to group constraints.
6	DEVICE_CONSTRAINT	The parameter list contains one or more parameters that are related to device constraints.
7	COUNTER_CONSTRAINT	The parameter list contains one or more parameters that are related to counter constraints.

In this table, the parameter list mentioned in the descriptions is the ‘Parameters’ field of the ExtendedStatusBlock data structure.

3.3.2.2 Describe Routines

<table>
<tr><th>Value</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>UNSPECIFIED</td><td>Unspecified (N/A)
Subcategories:
<table><tr><th>Id</th><th>Name</th><th>Description</th></tr><tr><td>0</td><td>UNSPECIFIED</td><td>Unspecified (N/A)</td></tr></table></td></tr>
</table>

For those categories, the same local flags as the ones defined for action routines apply.

Describe routines SHOULD include in their returned ESB a parameter named ‘Description’ as specified in §3.3.4.1.

Describe routines MUST NOT contain in their returned ESB any obligation or callback parameters.

Describe routines SHOULD return ESB parameters that describe some or all of the constraints that are applicable for the corresponding action or Link constraint.

3.3.2.3 Link Constraint Routines

<table>
<tr><th>Value</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>LINK_VALID</td><td>The Link constrained by this control program is valid.
The parameter list of the returned status block SHOULD NOT contain any of the constraint parameters, and MUST NOT contain obligation or callback parameters
Subcategories:
<table>
<tr><th>Id</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>UNSPECIFIED</td><td>Unspecified (N/A)</td></tr>
</table></td></tr>
<tr><td>1</td><td>LINK_INVALID</td><td>The Link constrained by this control program is invalid.
When a Link is invalid, the control program SHOULD include in the parameter list of the returned ESB one or more of the constraints that were not met and caused the Link to be invalid. (The constraints that were not evaluated and the constraints that did not cause the action to fail SHOULD be omitted.)
The parameter list of the returned ESB MUST NOT contain any obligation or callback parameter.
Subcategories:
<table>
<tr><th>Id</th><th>Name</th><th>Description</th></tr>
<tr><td>0</td><td>UNSPECIFIED</td><td>Unspecified (N/A)</td></tr>
</table></td></tr>
</table>

For those categories, the same local flags as the ones defined for action routines apply.

In the context of ESB parameters returned by Link constraint routines, a constraint means a condition that is required to be true or a criterion that is required to be met in order for the result of the routine to return an ESB with the Category LINK_VALID.

3.3.3 Cache Durations

The CacheDuration field of an ESB is an indication of the validity period of the information encoded in the ESB. When an ESB has a non-zero validity period, it means that the ESB can be stored in a cache, and that during that period of time, a call to the exact same routine with the same parameters would return the same ESB, so the cached value MAY be returned to the host application instead of calling the routine.

3.3.4 Parameters

Some parameters are used to convey detailed information about the return status, as well as variable bindings for template processing.

NOTE: Except for the obligations and callbacks, all the constraints described here are strictly for the purpose of helping the host application classify and display, not for enforcement of the usage rules. The enforcement of the rules is solely the responsibility of the control program.

The parameters defined in the sections below (3.3.4.1 through 3.3.4.2.6) are encoded either as a ParameterBlock (§4.7.3.1.1) if no parameter flags are applicable or as an

ExtendedParameterBlock (§4.7.3.1.2) if one or more flags are applicable. Flags are specified in §3.3.4.3.

3.3.4.1 Description

Parameter Name: Description

Parameter Type: ValueList

Description: List of description parameters. Each value in the list is of type Parameter or ExtendedParameter. The following parameters are defined: Default, Short and Long.

Each of them, if present, has for its value the ID of one of the control's resources. That resource SHOULD contain a textual payload, or a template payload (as described in §3.5). If the resource is a template, it is processed to obtain a textual description of the result (either a description of the entire control program, or of a specific action). The template is processed using as variable bindings the other parameters of the list in which the 'Description' parameter appears.

The 'Short' and 'Long' descriptions can only be included if a 'Default' description is also included.

Name	Type	Description
Default	Resource	ID of the resource that contains the normal description text or template
Short	Resource	ID of the resource that contains the short description text or template
Long	Resource	ID of the resource that contains the long description text or template

3.3.4.2 Constraints

Constraint parameters are grouped in lists that contain constraints of similar types. This specification defines standard constraints for some of the types. Controls MAY return constraint parameters that are not defined in this specification, provided that the name of each constraint parameter is a URN in a namespace that guarantees the uniqueness of that name. This MAY include vendor-specific constraints, or constraints defined in other specifications.

3.3.4.2.1 Generic Constraints

Parameter Name: GenericConstraints

Parameter Type: ValueList

Description: List of generic constraints that may be applicable. Each value in the list is of type Parameter or ExtendedParameter.

Generic constraints are constraints that do not belong to any of the other constraint types defined in §3.3.4.2

The following generic constraint parameters are defined:

Name	Type	Description

NodeReachabilityRequired	String	ID of a Node that is required to be reachable.

3.3.4.2.2 Temporal Constraints

Parameter Name: TemporalConstraints

Parameter Type: ValueList

Description: List of temporal constraints that may be applicable. Each value in the list is of type Parameter or ExtendedParameter.

Temporal constraints are constraints that are related to time, dates, durations, etc.

The following temporal constraint parameters are defined:

Name	Type	Description
NotBefore	Date	Date before which the action is denied.
NotAfter	Date	Date after which the action is denied.
NotDuring	ValueList	List of 2 values of type Date. The first value is the start of the period, and the second is the end of the period that is excluded.
NotLongerThan	Integer	Max number of seconds after first use. This value is unsigned.
NotMoreThan	Integer	Max number of seconds of accumulated use time. This value is unsigned.

3.3.4.2.3 Spatial Constraints

Parameter Name: SpatialConstraints

Parameter Type: ValueList

Description: List of spatial constraints that may be applicable. Each value in the list is of type Parameter or ExtendedParameter.

Spatial constraints are constraints that are related to physical locations.

No standard spatial constraint is defined in this specification.

3.3.4.2.4 Group Constraints

Parameter Name: GroupConstraints

Parameter Type: ValueList

Description: List of group constraints that may be applicable. Each value in the list is of type Parameter or ExtendedParameter.

Group constraints are constraints that are related to groups, group membership, identity groups, etc.

The following parameters are defined:

Name	Type	Description
MembershipRequired	Resource	ID of the resource that contains the text or template for the name or identifier of a group for which a membership is required
IdentityRequired	Resource	ID of the resource that contains the text or template for the name or identifier of an individual

3.3.4.2.5 Device Constraints

Parameter Name: DeviceConstraints

Parameter Type: ValueList

Description: List of device constraints that may be applicable. Each value in the list is of type Parameter or ExtendedParameter.

Device constraints are constraints that are related to characteristics of a device, such as features, attributes, names, identifiers, etc.

The following parameters are defined:

Name	Type	Description
DeviceTypeRequired	Resource	ID of the resource that contains the text or template for the type of host device that is required
DeviceFeatureRequired	Resource	ID of the resource that contains the text or template for the name of a feature that the host device must have
DeviceIdRequired	String	ID that the device is required to have. This ID MAY be any string that can be used to identify the device (device name, device serial number, a Node ID, etc.)

Obligation and Callback parameters are described in §3.4.

3.3.4.2.6 Counter Constraints

Parameter Name: CounterConstraints

Parameter Type: ValueList

Description: List of counter constraints that may be applicable. Each value in the list is of type Parameter or ExtendedParameter.

Counter constraints are constraints that are related to counted values, such as play counts, accumulated counts, etc.

The RepeatCount parameter is defined as follows:
There MUST be only one RepeatCount parameter.

Name	Type	Description
RepeatCount	ValueList	List of 2 unsigned Integer values. The first value is the number of repeats that must not be exceeded. The second value is the number of repeats that are still allowed.

For example, if you can repeat an action 3 times, the constraint would be:

RepeatCount = [3,3] before the first 'Perform'
RepeatCount = [3,2] after the first 'Perform'
RepeatCount = [3,1] after the second 'Perform'
RepeatCount = [3,0] after the third 'Perform'

The first value MAY be 0 if the maximum value is unknown. For example, this is used for a case where there is an item which allows playing a limited number of times, but there is no set maximum: first 'purchase' some number of plays, and afterwards purchase some additional number of plays. So, there is a count (it can say: "you can play this content X more times"), but there's no fixed maximum.

3.3.4.3 Parameter Flags

The following flags MAY be used for all the parameters described in §3.3.4 of this specification when they are encoded as an ExtendedParameterBlock.

Bit Index (0 is LSB)	Name	Description
0	CRITICAL	The semantics associated with this parameter need to be understood by the host application. If they are not, the entire ESB should be treated as not understood and rejected. This flag SHOULD NOT be used for parameters that are descriptive in nature.
1	HUMAN_READABLE	This parameter represents a value whose name and value are suitable to display in a textual or graphical user interface. Any parameter that does not have this flag set should be reserved for the host application, and not be shown to a user. For parameter values of type Resource, it is not the resource ID but the resource data payload referenced by the ID that is human-readable.

3.4 Obligations and Callbacks

Certain actions, when granted, require further participation from the host application. Obligations represent operations that need to be performed by the host application upon or after the use of the content key it is requesting. Callbacks represent calls to one or more of the control program routines that need to be performed by the host application upon or after the use of the content key they are requesting.

If an application encounters any critical obligation or callback that it does not support, or does not understand (for example because the obligation type may have been defined after the application was implemented), it MUST refuse to continue the action for which this obligation or callback parameter was returned. A critical obligation or callback is indicated by setting the CRITICAL parameter flag for the parameter that describes it.

If a control has side effects (such as decrementing a play count, for example), it SHOULD use the OnAccept callback to require the host application to call a certain routine if it is able to understand and comply with all critical obligations and callbacks. The side effect SHOULD happen in the callback routine. All implementations MUST understand and implement the OnAccept callback.

3.4.1 Parameters

The following parameters define several types of obligations and callbacks that can be returned in ExtendedStatusBlock data structures.

3.4.1.1 Obligations

Parameter Name: Obligations

Parameter Type: ValueList

Description: List of obligation parameters. Each value in the list is of type Parameter or ExtendedParameter. The following obligation parameters are defined:

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>RunAgentOnPeer</td><td>ValueList</td><td>The host application MUST send an agent control to run on the peer of the currently running protocol session.
<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>String</td><td>ID of the Control that contains the agent to run.</td></tr>
<tr><td>String</td><td>Name of the agent to run.</td></tr>
<tr><td>Integer</td><td>Instance ID. This value is used to uniquely identify this agent obligation instance. This ID will also allow the system to correlate this agent obligation with an OnAgentCompletion callback parameter.</td></tr>
<tr><td>String</td><td>Context ID. This ID will be visible to the running agent on the peer under the agent’s session context host object path: Octopus/Agent/Parameters/Session/Co</td></tr>
</table></td></tr>
</table>

			ntextId.
		ValueList	List of values of type Parameter. All these parameters will be visible to the agent as input parameters. These agent parameters MUST NOT contain values for which there are not defined mappings to host objects.

3.4.1.2 Callbacks

Parameter Name: Callbacks

Parameter Type: ValueList

Description: List of callback parameters. Each value in the list is of type Parameter or ExtendedParameter. The following callback parameters are defined:

Name	Type	Description
OnAccept	Callback	The host application MUST call back if it is able to comply with all the requirements signaled in the ESB unless the host application can determine that it cannot successfully perform the action (see NOTE 1 below). The host application MUST NOT call back if it is unable to comply with all the requirements signaled in the ESB. The requirements signaled in the ESB include any and all callbacks and obligations that cannot be ignored (they are marked as CRITICAL, as defined in §3.3.4.3) and any other implicit or explicit requirements (e.g., permissions) indicated by other fields of the ESB. There MUST be at most one OnAccept callback parameter in a list of callback parameters. If other callback parameters are specified in the list, the OnAccept callback MUST be executed first. NOTE 1: This callback is likely to trigger some type of side effect. If the host application can determine that it is unable to successfully perform the action, it MUST NOT call back. It is understood that host applications are not always able to determine, for all actions, whether they can be successfully performed. Such a determination is implementation specific.

<table>
<tr><td>OnTime</td><td>ValueList</td><td>The host application MUST call back after the specified date/time has passed.
<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>Date</td><td>The date/time after which the host application needs to perform the callback.</td></tr>
<tr><td>Callback</td><td>Routine to call back, and associated cookie.</td></tr>
</table></td></tr>
<tr><td>OnTimeElapsed</td><td>ValueList</td><td>The host application MUST call back after the specified duration has elapsed. (The counting starts when the host application actually performs the action for which the permission was granted.)
<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>Integer</td><td>Number of seconds. The value is unsigned.</td></tr>
<tr><td>Callback</td><td>Routine to call back, and associated cookie.</td></tr>
</table></td></tr>
<tr><td>OnEvent</td><td>ValueList</td><td>The host application MUST call back when a certain event occurs.
<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>String</td><td>Event Name.</td></tr>
<tr><td>Integer</td><td>Event Flags. (The integer value is interpreted as a bit-field.)</td></tr>
<tr><td>Integer</td><td>Event Parameter.</td></tr>
<tr><td>Callback</td><td>Routine to call back, and associated cookie.</td></tr>
</table>
See the section about events, §3.3.4.2, for more details about the events.</td></tr>
</table>

<table>
<tr>
<td>OnAgentCompletion</td>
<td>ValueList</td>
<td>
<p>The host application MUST call back when an agent specified in one of the obligation parameters has completed, or failed to run.</p>
<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>Integer</td><td><p>Agent instance ID.</p><p>The instance ID specified in an agent obligation.</p></td></tr>
<tr><td>Callback</td><td>Routine to call back, and associated cookie.</td></tr>
</table>
<p>When calling back, the host application MUST provide the following ArgumentsBlock:</p>
<table>
<tr><th>Type</th><th>Encoding</th><th>Description</th></tr>
<tr><td>32-bit integer</td><td>4 bytes in big-endian order</td><td>Completion status code.</td></tr>
<tr><td>32-bit integer</td><td>4 bytes in big-endian order</td><td>Agent result code.</td></tr>
<tr><td>8-bit byte array</td><td>Byte sequence</td><td>Agent ReturnBlock.</td></tr>
</table>
<p>The completion status code value is 0 if the agent was able to run, or a negative error code if it was not.</p>
<p>The agent ReturnBlock is the data returned by the agent. This is omitted if the agent was unable to run (that is, if the completion status code is not 0).</p>
</td>
</tr>
</table>

The ‘Callback’ type mentioned in the table above is a ValueListBlock with three ValueBlock elements.

Value Type	Description

<table>
<tr><td>Integer</td><td>ID of the callback type. Two types of callbacks are defined:
<table>
<tr><th>ID</th><th>Description</th></tr>
<tr><td>RESET = 0</td><td>All pending callback requests and active obligations are cancelled upon calling the callback routine. The callback routine returns an ESB that indicates if and how the application can continue with the current operation.</td></tr>
<tr><td>CONTINUE =1</td><td>The callback routine is called while all other pending callback requests and active obligations remain in effect. The callback routine returns a simple result code. The application can continue with the current operation unless that result code indicates a failure.</td></tr>
</table></td></tr>
<tr><td>String</td><td>Entry point to call in the code module. This MUST be one of the entries in the Export Table of the code module for the same Control as the one containing the routine that returned the ESB with this parameter.</td></tr>
<tr><td>Integer</td><td>Cookie. This value will be passed on the stack to the routine that is called.</td></tr>
</table>

3.4.1.3 Parameter Flags

The same parameter flags as those defined in §3.3.4.3 are applicable. It is important that all callbacks and obligations that the caller is required to implement be marked as CRITICAL; otherwise, a host application MAY have the choice to ignore those parameters.

3.4.2 Events

Events are specified by name. Depending on the type of event, there may be a set of flags defined that further specify the event; if no flags are defined for a specific event, the value of the Flag field MUST be 0. Also, some events may specify that some information be supplied to the callback routine when the event occurs; if no special information is required from the host application, the application MUST call with an empty ArgumentsBlock (see the description of the callback routine interface in §3.4.3).

If the name of an event in a callback parameter marked CRITICAL is not understood or not supported by the host application, the application MUST consider this parameter as a not-understood CRITICAL parameter; the action for which permission was requested MUST NOT be performed.

The following event names are defined:

Event Name	Event Flags	Event Parameter	Description
OnPlay	None	None	The host application MUST call back when the multimedia object starts playing.
OnStop	None	None	The host application MUST call back when the multimedia stops playing (or is paused).

<table>
<tr><td>OnTimecode</td><td>None</td><td>Presentation time expressed in number of seconds since the start of the presentation</td><td>The host application MUST call back when the specified presentation time has been reached or exceeded (during normal real-time playback or after a seek). The origin of the presentation time is when the rendering begins. The presentation time relates to the source media time, not the wall-clock time; when a presentation is paused, the presentation time does not change.</td></tr>
<tr><td>OnSeek</td><td>None</td><td>None</td><td><p>The host application MUST call back when a direct access to an arbitrary point in a multimedia presentation occurs.</p><p>When calling back, the host application MUST provide the following data in an ArgumentsBlock:</p><table><tr><th>Type</th><th>Encoding</th><th>Description</th></tr><tr><td>32-bit unsigned integer</td><td>4 bytes in big-endian order</td><td>Seek position offset</td></tr><tr><td>32-bit unsigned integer</td><td>4 bytes in big-endian order</td><td>Seek position range</td></tr></table><p>The position within the multimedia presentation is <i>offset</i> ‘marks’ out of <i>range</i> total ‘marks’ in the presentation.</p><p>For instance, for a presentation that is 327 seconds long, seeking to the 60th second can be represented with offset=60, range=327. It is up to the caller to choose the unit that corresponds to the measurement of the offset and range (it could be a time unit, a byte-size unit, or any other unit), provided that the ‘marks’ are homogeneously distributed over the entire presentation. The value of <i>offset</i> MUST be less than or equal to the value of <i>range</i>.</p></td></tr>
</table>

3.4.3 Callback Routines

All callback routines take the same input:

Input:

Top of stack:

Cookie
ArgumentsBlockSize
…data…

Cookie: The value of the Cookie field that was specified in the callback parameter.

ArgumentsBlockSize: The number of bytes of data passed on the stack below this parameter. When the routine is called, the stack contains the value ArgumentsBlockSize supplied by the caller (indicating the size of the arguments block) at the top, followed by ArgumentsBlockSize bytes of data. If the size is not a multiple of 4, the data on the stack will be padded with 0-value bytes to ensure that the Stack Pointer remains a multiple of 4.

3.4.3.1 CONTINUE Callbacks

Callbacks with the type CONTINUE have the following output:

Output:

Top of stack:

ResultCode
…

ResultCode: An integer value. The result value is 0 if the routine was able to execute, or a negative error code if an error occurred.

Description: If the ResultCode indicates that the callback routine was able to run (value 0), the host application can continue the current operation. If the ResultCode indicates that an error occurred, the host application MUST abort the current operation and cancel all pending callbacks and obligations.

3.4.3.2 RESET Callbacks

When a control routine has specified one or more callbacks of type RESET in the ESB returned from a routine, the host application SHALL call any specified callback routine when the condition for that callback is met. As soon as the conditions of any of the callbacks are met, the host application MUST:

- Cancel all other pending callbacks
- Cancel all current obligations
- Provide the required parameters (if any) for that callback
- Call the specified callback routine

The return status from the routine indicates to the host application whether it can continue performing the current operation. If the permission is denied or the routine fails to execute successfully, the host application MUST abort the performance of the current operation. If the permission is granted, the host application MUST comply with any obligation or callback that may be returned in an ESB, just as if it had called the original Control.Actions.<Action>.Perform routine. Previous obligations or callback specifications are no longer valid.

All routines specified as callback entry points for this type of callback have the following output:

Output:

Top of stack:

ResultCode

StatusBlockPointer
…

ResultCode: An integer value. The result value is 0 if the routine was able to execute, or a negative error code if an error occurred.

StatusBlockPointer: Address of a standard ExtendedStatusBlock.

Description: The return semantics of this routine are equivalent to what is described for the Control.Actions.<Action>.Perform routine.

3.5 *Metadata Resources*

Control objects can contain metadata resources, which can be referenced from the parameters returned in ExtendedStatusBlock data structures. Resources can be simple text, text templates, or other data types. Each resource is identified by a resource ID, and can contain one or more text strings or encoded data, one for each version in a different language. It is not required that resources be provided for all languages. It is up to the host application to choose which language version is most appropriate for its needs.

An implementation MAY choose to not dereference the content of resources if it does not need to use the data in the resource. For example, a device without a user interface is likely to not need to dereference resources referenced from the 'Description' parameters in an ExtendedStatusBlock.

Resource		
Field	**Type**	**Description**
Id	ASCII String	ID of a resource in a ResourceList extension
Type	ASCII String	MIME-type of the resource data, as described in IETF RFC 2046 [RFC2046]
Data	List of LocalizedData	List of all the different versions of the resource, for different locales

LocalizedData		
Field	**Type**	**Description**
Language	ASCII String	Language code, as specified in IETF RFC 3066 [RFC3066]
Data	Type depends on the specified mime type	The actual data for the resource (text, etc.)

Resources accompany control programs by being included in a ResourceList Extension in a Control object. The resource ID maps to the ID of a Resource in a ResourceList internal extension of the Control object that contains the code module with the routine that is currently running.

For the purpose of computing the canonical byte sequence for Resource objects, the data structure description is the following (see §2 for more details on canonical byte sequences):

```
class LocalizedData {
    string language
    byte[] data;
}

class Resource {
    string id
    string type;
    LocalizedData[] data;
}

class ResourceList {
    Resource[] resources;
}

class ResourceListExtension extends ExtensionData(type=’ResourceList’) {
    ResourceList resources;
}
```

3.5.1 Simple Text

Simple text is specified as MIME-type ‘text’.

3.5.2 Text Templates

In addition to the standard text resources, this specification defines a text template type. The MIME-type for this is ‘text/vnd.intertrust.octopus-text-template’.

A text template contains text characters encoded in UTF-8, as well as named placeholders that are to be replaced by text values obtained from parameters returned in the parameters list, such as that of an ExtendedStatusBlock. The syntax for a placeholder is ‘\PLACEHOLDER\’, where PLACEHOLDER specifies the name of a ParameterBlock and an OPTIONAL formatting hint. The template processor MUST replace the entire token ‘\PLACEHOLDER\’ with the formatted representation of the Value field of that ParameterBlock. The formatting of the Value data is specified in §3.5.2.1.

If the character ‘\’ appears in the text outside of a placeholder, it MUST be encoded as ‘\\’. All occurrences of ‘\\’ in the text SHALL be reverted to ‘\’ by the template processor.

The syntax for the placeholder is: FORMAT|NAME, where NAME is the name of a ParameterBlock, and FORMAT is the formatting hint to convert the parameter’s data into text. If the default formatting rules for the parameter’s value’s data type are sufficient, then the formatting hint can be omitted, and the placeholder is simply NAME.

3.5.2.1 Formatting

3.5.2.1.1 Default Formatting

The default formatting rules for the different value types are:

Type	Formatting

Integer	Text representation of the integer value as a signed decimal. The text is composed only of the characters for the digits “0” to “9” and the character “-“. If the value is 0, the text is the string “0”. If the value is not 0, the text does not start with the character “0”. If the value is negative, the text starts with the character “-”. If the value is positive, the text starts with a non-zero digit character.
Real	Text representation of the floating point value in decimal. The integral part of the value is represented using the same rules as for Integer values. The decimal separator is represented with the host application’s preferred decimal separator. The fractional part of the value consists of up to 6 “0” characters followed by up to 3 non-zero digit characters.
String	The string value itself.
Date	A human-readable representation of the date, according to the host’s preferred text representation of dates.
Parameter	The text “<name>=<value>”, where <name> is the parameter name, and <value> is the parameter value formatted according to the default formatting rules for its type.
ExtendedParameter	Same as for Parameter.
Resource	Text string of the resource’s data. The resource referenced by the placeholder MUST have a MIME-type that is text-based (e.g., text or text/vnd.intertrust.octopus-text-template).
ValueList	The text “<value>, <value>, …” with all the values in the list formatted according to the default formatting rules for their type.

3.5.2.1.2 Explicit Formatting

Those format names can be used as the FORMAT part of a placeholder tag. If an unknown FORMAT name is encountered, the template processing engine will use the default formatting rules.

Name	Formatting
Hex	Hexadecimal representation of an integer value interpreted as unsigned. NOTE: This formatting hint should be ignored for data types that are not integers.

3.6 *Context Objects*

When a control routine is executing, it has access to a number of context objects, through the use of the System.Host.GetObject system call (§4.7.2.7).

3.6.1 General Context

This Context MUST be present for all running controls.

Name	Type	Description

Octopus/Personality/Id	String	ID of the current Personality Node
Octopus/Personality/Attributes	Container of Attributes	Attributes of the current Personality Node

3.6.2 Runtime Context

This context MUST be present for all running controls running in a VM that has been created using the System.Host.SpawnVm system call. For controls running in a VM not created using System.Host.SpawnVm, this context MUST be nonexistent, or an empty container.

Name	Type	Description
Octopus/Runtime/Parent/Id	Container of unnamed String objects	The identity under which the caller of the system call is running. The Id value is an array of names, one for each of the names associated with this identity.

3.6.3 Control Context

This context MUST be present whenever a routine of a control is running.

Name	Type	Description
Octopus/Control/Id	String	ID of the running Control
Octopus/Control/Attributes	Container	Attributes of the running control. This object MAY be omitted if the control has no attributes.

3.6.4 Controller Context

This Context MUST be present whenever a routine of a control is running and the control was pointed to by a Controller object (e.g., when accessing a ContentKey object when consuming content).

It is guaranteed that for any action there will be only one applicable Controller object, because of the restriction imposed on the system to only allow a host application to group content keys that are controlled by a single Controller object.

Name	Type	Description
Octopus/Controller/Id	String	ID of the Controller that points to the currently running Control.
Octopus/Controller/Attributes	Container	Attributes of the Controller pointing to the currently running control. This object MAY be omitted if the controller has no attributes.

3.6.5 Action Context

This Context MUST be present whenever a control is called for the purpose of controlling an action.

Name	Type	Description
Octopus/Action/Parameters	Container	Array of Name/Value pairs representing the parameters that are relevant for the current action, if any. Each action type defines a list of OPTIONAL and REQUIRED parameters. This container MAY be omitted if the action has no parameters.

3.6.6 Link Context

This context MUST be present whenever a control is called for the purpose of limiting the validity of a Link object (i.e., the Control object is embedded in a Link object).

Name	Type	Description
Octopus/Link/Id	String	ID of the Link object
Octopus/Link/Attributes	Container	Attributes of the Link object that contains the running Control. This object MAY be omitted if the Link has no attributes.

3.6.7 Agent Context

This context MUST be present whenever an agent routine of a control is running.

Name	Type	Description
Octopus/Agent/Parameters	Container	Array of Name/Value parameter pairs representing the input parameters for the agent.
Octopus/Agent/Session/ContextId	String	Identifier for the session context in which the agent is running.

The Parameters and Session containers are normally used to allow the protocols that require one entity to send and run an agent on another entity to specify which input parameters to pass to the agent, and which session context objects the host needs to set under certain conditions. The presence or absence of certain session context objects may allow the agent code to decide whether it is running as part of the protocol it was designed to support, or if it is running out of context, in which case it may refuse to run. For example, an agent whose purpose is to create a state object on the host on which it runs may refuse to run unless it is being executed during a specific protocol interaction.

3.7 *Actions*

Each action has a name, and a list of parameters. Some parameters are REQUIRED (the application MUST provide them when performing this action), and some are OPTIONAL (the application MAY provide them, or MAY omit them).

The following standard actions are defined:

3.7.1 Play

Description: Normal real-time playback of the multimedia content.

Parameters:

Name	Type	Description
N/A	N/A	N/A

3.7.2 Transfer

Description: Transfer to a compatible target system.

Transferring to a compatible target system is used when the content has to be made available to a system with the same DRM technology, such that the target system can use the same license as the one that contains this control, but state information may need to be changed on the source, the sink, or both. The system from which the transfer is being done is called the source. The target system to which the transfer is being done is called the sink.

This action is intended to be used in conjunction with a service protocol that allows an agent to be transferred from the source to the sink in order to do the necessary updates in the source’s and sink’s persistent states (objects in a SeaShell database, see §7). A control uses the RunAgentOnPeer obligation for that purpose.

Parameters:

Name	Type	Description
Sink/Id	String	Octopus Node ID of the Sink.
Sink/Attributes	Container	Attributes of the Sink’s Octopus Node. This container MAY be omitted if the Node has no attributes.

<table>
<tr><td>TransferMode</td><td>String</td><td>Transfer Mode ID indicating in which mode the content is being transferred. This ID can be a standard mode as defined below, or a URN for a system-proprietary mode.
The following standard modes are defined:
<table>
<tr><th>ID</th><th>Description</th></tr>
<tr><td>Render</td><td>The sink is receiving the content for the purpose of rendering.</td></tr>
<tr><td>Copy</td><td>The sink is receiving a copy of the content.</td></tr>
<tr><td>Move</td><td>The content is being moved to the sink.</td></tr>
<tr><td>CheckOut</td><td>The content is being checked-out to the sink. This is similar to Move but with the distinction that the resulting state on the sink may prevent any other move than a move back to the source.</td></tr>
</table></td></tr>
<tr><td>TransferCount</td><td>Integer</td><td>Integer value indicating how many instances of the state counters associated with this control need to be transferred to the sink.
This parameter is OPTIONAL. If it is not present, only one instance is being transferred. It should not be present when the transfer mode is Render or Copy.</td></tr>
</table>

3.7.3 Export

Description: Export to a foreign target system.

Exporting to a foreign target system is an action that is used when the content has to be exported to a system where the original content license cannot be used. This could be a system with a different DRM technology, a system with no DRM technology, or a system with the same technology but under a situation that requires a license different from the original license. The system from which the transfer is being done is called the source. The target system to which the transfer is being done is called the sink.

In the Extended Status result for the Describe, Check and Perform methods of this action, the following parameter shall be set:

Name	Type	Description
ExportInfo	Any	Information that is relevant when exporting content to the target system specified in the action parameters. The actual type and content of this information is specific to each target system. For example, for CCI-based systems, this would contain the CCI bits to set for the exported content.

Parameters:

Name	Type	Description

<table>
<tr><td>TargetSystem</td><td>String</td><td>System ID of the foreign system to which the export is being made. This ID is a URN.</td></tr>
<tr><td>ExportMode</td><td>String</td><td>Export Mode ID indicating in which mode the content is being exported. This ID can be a standard mode as defined below, or a URN for a system-proprietary mode.
The following standard modes are defined:
<table>
<tr><th>ID</th><th>Description</th></tr>
<tr><td>DontKnow</td><td>The caller does not know what the sink’s intended mode is. In this case, the control program should assume that any of the allowed modes for the TargetSystem can be assumed by the sink, and should indicate any restriction in the return status of the action routines. For example, for a CCI-based system, the control can return CCI bits that will either allow the equivalent of Render or Copy depending on what the license permits.</td></tr>
<tr><td>Render</td><td>The sink is receiving the content for the purpose of rendering, and will not retain a usable copy of the content except for caching purposes as specified by each target system.</td></tr>
<tr><td>Copy</td><td>The sink is receiving a copy of the content.</td></tr>
<tr><td>Move</td><td>The content is being moved to the sink.</td></tr>
</table></td></tr>
</table>

Other input parameters may be required by specific target systems.

3.7.3.1 Standard Target Systems

3.7.3.1.1 Audio CD or DVD

The standard TargetSystem ID ‘CleartextPcmAudio’ is used when the target system is an un-encrypted medium onto which uncompressed PCM audio is written, such as a writeable audio CD or DVD.

For this target system, the ExportInfo parameter is a single Integer parameter representing a copyright flag. This flag is indicated in the least significant bit of the integer value.

Bit index	Description
0 (LSB)	When this flag is set, the Copyright bit or flag MUST be set in the format of the recoded audio if the format supports the signaling of such a bit or flag.

4 Plankton Virtual Machine

4.1 *Introduction*

This section of the document provides the specification for Plankton, the virtual machine (VM) used by the Octopus Engine to execute control programs that govern access to content.

In this specification, we explain some of the design decisions and where the Plankton VM fits in the Octopus architecture, then describe the basic elements of the VM, followed by more details about the memory model and instruction set. We then describe how programs are packaged in code modules. Finally, we document the system calls available to programs.

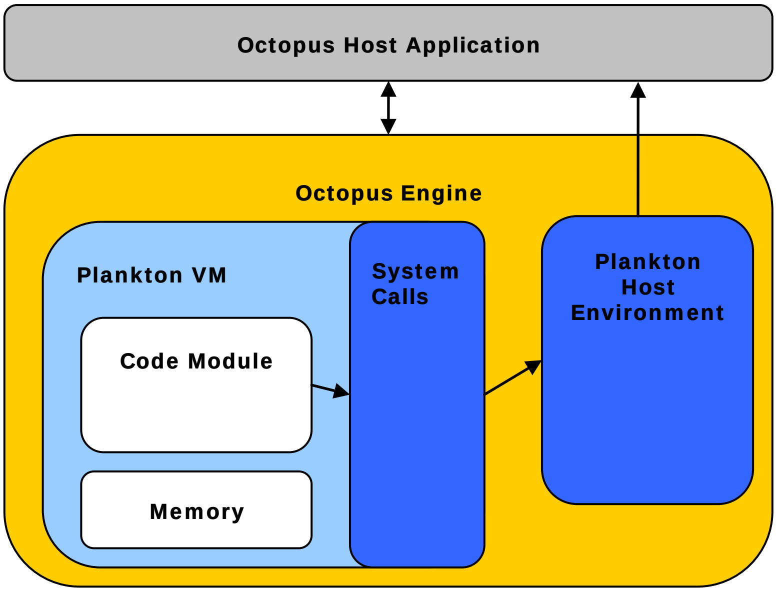
4.2 *Design Rationale*

The Plankton Virtual Machine (VM) is a traditional virtual machine, designed to be easy to implement using various programming languages, with a very small code footprint. It is based on a stack-oriented instruction set that could be called a TISC (Trivial Instruction Set Computer) architecture. The instruction set is designed to be minimalist, without much concern for execution speed or code density. Execution speed is a non-goal, and code density is an orthogonal problem: when compact code is required, data compression techniques will be used to compress the Plankton byte code, instead of making the byte code compact by design.

The Plankton VM should be suitable as a target for both low-level and high-level programming languages. At a minimum, the virtual machine should naturally support Assembler, C and FORTH. It should be possible to implement compilers for other languages, such as Java or custom languages, without too much trouble.

Finally, the Plankton VM is designed to be hosted within a host environment, not run directly on a processor or in silicon. The natural host environment for Plankton is the Octopus Engine.

4.3 *Architecture*



The Plankton VM runs within the context of its host environment, which implements some of the functions needed by the VM as it executes programs. Typically, the Plankton VM runs within the Octopus Engine, which implements its host environment.

The VM runs programs by executing instructions stored in byte codes in code modules. Some of these instructions can call functions implemented outside of the program itself by making a *system call*. System calls are either implemented by the Plankton VM itself, or delegated to the host environment.

4.4 *Basic VM Elements*

4.4.1 Execution Model

The Plankton VM executes instructions stored in code modules as a stream of byte codes loaded in memory. The VM maintains a virtual register called the Program Counter (PC), which is incremented as instructions are executed. The VM executes each instruction, in sequence, until the `OP_STOP` instruction is encountered, an `OP_RET` instruction is encountered with an empty call stack, or a runtime exception occurs. A jump is specified either as a relative jump (specified as a byte offset from the current value of PC), or an absolute address.

4.4.2 Memory Model

The Plankton VM has a simple memory model. The VM memory is separated into the Data Memory Space and the Code Memory Space.

The Data Memory is a single, flat, contiguous memory space, starting at address 0. The Data Memory is typically implemented as an array of bytes allocated within the heap memory of the host application or host environment. Any attempt to access memory outside of that space will cause a runtime exception that will cause the program execution to terminate. The data in the Data Memory can be accessed by memory-access instructions, which can be either 32-bit or 8-bit accesses. 32-bit memory accesses are done using the big-endian byte order. No assumptions are made with regards to alignment between the VM-visible memory and the host-managed memory (host CPU virtual or physical memory).

The Code Memory is a flat, contiguous memory space, starting at address 0. The Code Memory is typically implemented as an array of bytes allocated within the heap memory of the host application or host environment.

The VM MAY support loading more than one code module. If the VM loads several code modules concurrently, all the code modules share the same Data Memory (however, each module’s data is loaded at a different address), but each has its own Code Memory. This means that it is not possible for a jump instruction from one code module to cause a jump directly to code from another code module.

4.4.3 Data Stack

The VM has the notion of a Data Stack, which represents 32-bit data cells stored in the Data Memory. The VM maintains a virtual register called the Stack Pointer (SP). After reset, SP points to the end of the Data Memory, and the stack grows downward (when data is pushed on the Data Stack, the SP register is decremented). The 32-bit data cells on the stack are interpreted either as 32-bit addresses or 32-bit integers, depending on the instruction referencing the stack data. Addresses are unsigned integers. Unless otherwise specified, this specification considers all other 32-bit integer values on the Data Stack to be interpreted as signed integers.

4.4.4 Call Stack

The VM manages a Call Stack used for making subroutine calls. The values pushed on this stack cannot be read or written directly by any of the memory-access instructions. This stack is used internally by the VM when executing `OP_JSR`, `OP_JSRR` and `OP_RET` instructions. For a given VM implementation, the size of this return address stack will be fixed to a maximum, which will allow a certain number of nested calls that cannot be exceeded.

4.4.5 Pseudo-registers

The VM reserves a small address space at the beginning of the Data Memory to map pseudo-registers. The memory addresses of those pseudo-registers are fixed.

Address	Size	Name	Description
0	4	ID	32-bit ID of the currently executing Code Segment. This ID is chosen by the VM when a module is loaded. The VM will change this register if it changes from the Code Segment of one module to the Code Segment of another module.
4	4	DS	32-bit value set to the absolute data address at which the Data Segment of the currently executing module has been loaded. This value is determined by the VM’s module loader.
8	4	CS	32-bit value set to the absolute code address at which the Code Segment of the currently executing module has been loaded. This value is determined by the VM’s module loader.
12	4	UM	32-bit value set to the absolute data address of the first byte following the region of the Data Memory space where the Data Segments of code modules have been loaded.

4.4.6 Memory Map

The following shows the layout of the Data Memory and Code Memory spaces.

4.4.6.1 Data Memory

Address Range	Description
0 to 15	Pseudo-registers
16 to 127	Reserved for future VM/System use
128 to 255	Reserved for application use

256 to DS-1	Unspecified. The VM MAY load the Data Segments of code modules at any address at or above 256. If it chooses an address larger than 256, the use of the address space between 256 and DS is left unspecified. This means that the VM implementation is free to use it any way it sees fit.
DS to UM-1	Image of the Data Segments of one or more code modules loaded by the VM.
UM to End	Shared address space. The code modules' data and the Data Stack share this space. The Data Stack is located at the end of that space and grows down. End represents the last address of the data memory space. The size of the data memory space is fixed by the VM based on memory requirements contained in the code module and implementation requirements.

4.4.6.2 Code Memory

Address Range	Description
0 to CS-1	Unspecified. The VM MAY load the Code Segments of code modules at any address at or above 0. If it chooses an address larger than 0, the use of the address space between 0 and CS is left unspecified. This means that the VM implementation is free to use it any way it sees fit.
CS to CS+size(Code Segment)-1	Image of the Code Segment of a code module loaded by the VM.

4.4.7 Executing Routines

Before executing a code routine, a VM implementation MUST reset the Data Stack Pointer to point to the top of the initialized Data Stack. The initialized Data Stack consists of the routine's input data, and extends to the end of the Data Memory. The initialized Data Stack MAY be used as a way to pass input arguments to a routine. When there is no initialized Data Stack, the Data Stack Pointer points to the end of the Data Memory. The initial Call Stack MUST be either empty or contain a single terminal return address pointing to an `OP_STOP` instruction, which will force the execution or the routine to end on an `OP_STOP` instruction in case the routine finished with an `OP_RET` instruction.

When the execution stops, either because a final `OP_RET` instruction with an empty call stack has been executed or a final `OP_STOP` instruction has been executed, any data left on the Data Stack is considered to be the output of the routine.

4.4.8 Runtime Exceptions

When executing instructions, any of the following conditions is considered to be a runtime exception, and MUST cause the execution to stop immediately:

- An attempt to access data memory outside the current Data Memory address space.

- An attempt to set the PC to, or cause the PC to reach a code address outside the current Code Memory address space.
- An attempt to execute an undefined byte code.
- An attempt to execute an `OP_DIV` instruction with a top-of-stack operand equal to 0.
- An attempt to execute an `OP_MOD` instruction with a top-of-stack operand equal to 0.
- An overflow or underflow of the Call Stack.

4.5 *Instruction Set*

The Plankton VM uses a very simple instruction set. The number of instructions is very limited, but is sufficient to express programs of arbitrary complexity. Instructions and their operands are represented by a stream of byte codes. The instruction set is stack-based: except for the `OP_PUSH` instruction, none of the instructions have any direct operands. All operands are read from the Data Stack, and results pushed on the Data Stack. The VM is a 32-bit VM: all the instructions operate on 32-bit stack operands, representing either memory addresses or signed integers. Signed integers are represented with 2's complement binary encoding.

In the following table, the stack operands for instructions with two operands are listed as A,B with the top-of-stack last (B). Unless otherwise specified, the 'push' will mean pushing a 32-bit value as the new top cell on the Data Stack.

OP CODE	Name	Byte Code	Operands	Description
`OP_NOP`	No Operation	`0`		Do Nothing
`OP_PUSH`	Push Constant	`1`	`N (direct)`	Push a 32-bit constant
`OP_DROP`	Drop	`2`		Remove the top cell of the Data Stack
`OP_DUP`	Duplicate	`3`		Duplicate the top cell of the Data Stack
`OP_SWAP`	Swap	`4`		Swap top two stack cells
`OP_ADD`	Add	`5`	`A, B`	Push the sum of A and B (A+B)
`OP_MUL`	Multiply	`6`	`A, B`	Push the product of A and B (A*B)
`OP_SUB`	Subtract	`7`	`A, B`	Push the difference between A and B (A-B)
`OP_DIV`	Divide	`8`	`A, B`	Push the division of A by B (A/B)
`OP_MOD`	Modulo	`9`	`A, B`	Push A modulo B (A%B)
`OP_NEG`	Negate	`10`	`A`	Push the 2's complement negation of A (-A)

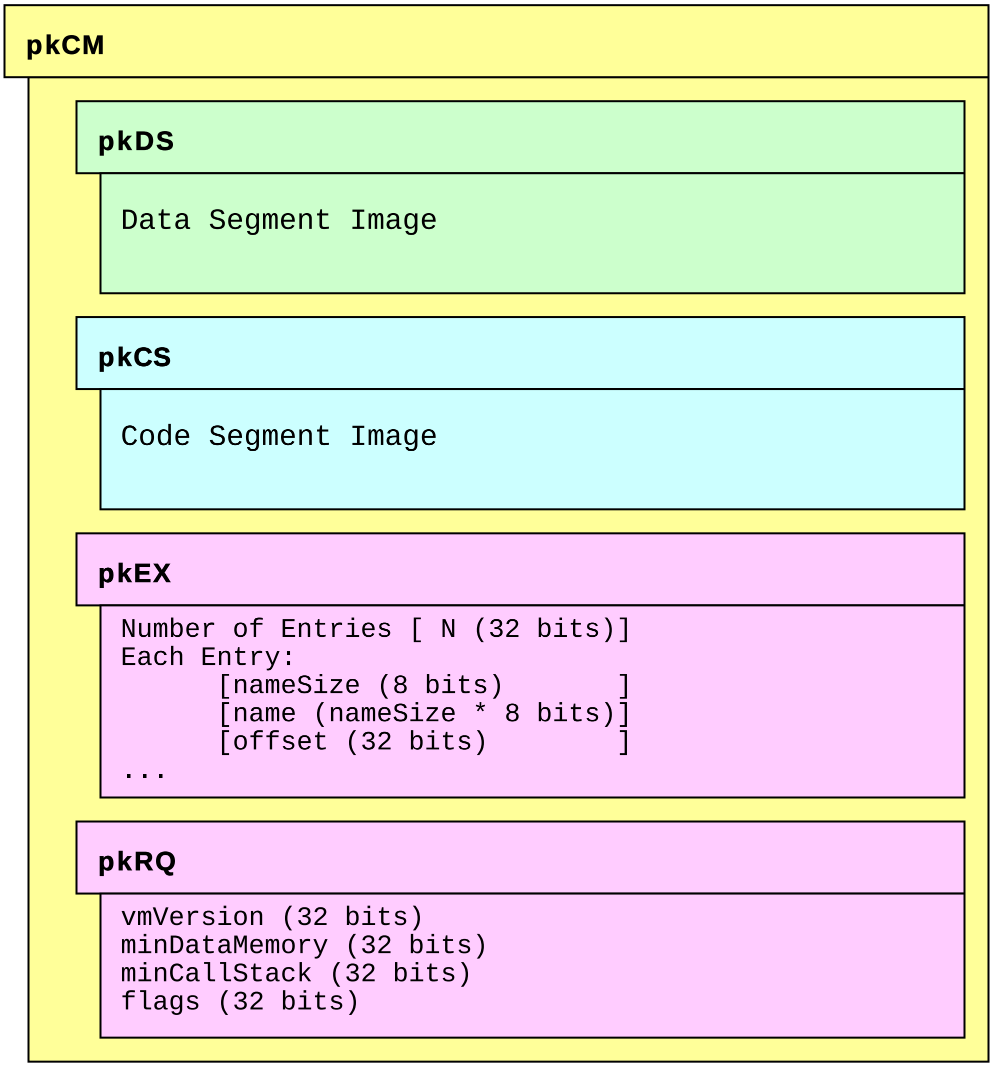
`OP_CMP`	Compare	`11`	`A, B`	Push -1 if A less than B, 0 if A equals B, and 1 if A greater than B
`OP_AND`	And	`12`	`A, B`	Push bit-wise AND of A and B (A & B)
`OP_OR`	Or	`13`	`A, B`	Push the bit-wise OR of A and B (A | B)
`OP_XOR`	Exclusive Or	`14`	`A, B`	Push the bit-wise eXclusive OR of A and B (A ^ B)
`OP_NOT`	Logical Negate	`15`	`A`	Push the logical negation of A (1 if A is 0, and 0 if A is not 0)
`OP_SHL`	Shift Left	`16`	`A, B`	Push A logically shifted left by B bits (A << B)
`OP_SHR`	Shift Right	`17`	`A, B`	Push A logically shifted right by B bits (A >> B)
`OP_JMP`	Jump	`18`	`A`	Jump to A
`OP_JSR`	Jump to Subroutine	`19`	`A`	Jump to subroutine at absolute address A. The current value of PC is pushed on the call stack.
`OP_JSRR`	Jump to Subroutine (Relative)	`20`	`A`	Jump to subroutine at PC+A. The current value of PC is pushed on the call stack.
`OP_RET`	Return from Subroutine	`21`		Return from subroutine to the return address popped from the call stack.
`OP_BRA`	Branch Always	`22`	`A`	Jump to PC + A
`OP_BRP`	Branch if Positive	`23`	`A, B`	Jump to PC+B if A > 0
`OP_BRN`	Branch if Negative	`24`	`A, B`	Jump to PC+B if A < 0
`OP_BRZ`	Branch if Zero	`25`	`A, B`	Jump to PC+B if A is 0
`OP_PEEK`	Peek	`26`	`A`	Push the 32-bit value stored at address A
`OP_POKE`	Poke	`27`	`A, B`	Store the 32-bit value A at address B
`OP_PEEKB`	Peek Byte	`28`	`A`	Read the 8-bit value stored at address A, 0-extend it to 32-bits and push it on the Data Stack
`OP_POKEB`	Poke Byte	`29`	`A, B`	Store the least significant 8 bits of value A at address B
`OP_PUSHSP`	Push Stack Pointer	`30`		Push the value of SP

`OP_POPSP`	Pop Stack Pointer	31	A	Set the value of SP to A
`OP_CALL`	System Call	32	A	Perform system call with index A
`OP_STOP`	Stop	255		Terminate Execution

4.6 *Code Modules*

4.6.1 Module Format

Code modules are stored in an atom-based format. Atoms are equivalent to the atom structure used in the MPEG-4 File Format: an atom consists of a 32-bit size represented by 4 bytes in big-endian byte order, followed by a 4-byte type (usually bytes that correspond to ASCII values of letters of the Alphabet), followed by the payload of the atom (size-8 bytes).



4.6.2 pkCM Atom

The pkCM atom is the top-level code module atom. It contains a sequence of sub-atoms. This atom MUST contain exactly one pkDS, one pkCS and one pkEX atom. It MAY contain one pkRQ atom. It MAY also contain any number of other atoms that MUST be ignored if present. The order of the sub-atoms is not specified, so an implementation MUST NOT assume a specific order.

4.6.3 pkDS Atom

The pkDS atom contains a memory image of a Data Segment that can be loaded into the Data Memory. The memory image is represented by a sequence of bytes. The memory image consists of one header byte, followed by zero or more data bytes. The header byte encodes a version number that identifies the format of the following data bytes.

Currently, only DataSegmentFormatVersion=0 is defined. In that format, the data bytes of the memory image represent a raw image to be loaded in memory. The VM loader only loads the data bytes of the memory image, not including the header byte. The VM loader MUST refuse to load an image in any other format version.

0	DS_0	DS_1	…

4.6.4 pkCS Atom

The pkCS atom contains a memory image of a Code Segment that can be loaded into the Code Memory. The memory image is represented by a sequence of bytes. The memory image consists of one header byte, followed by zero or mode data bytes. The header byte encodes a version number that identifies the format of the following data bytes.

Currently, only CodeSegmentFormatVersion=0 is defined. In that format the next byte contains a header byte representing a version number that identifies the byte code encoding for the following bytes. ByteCodeVersion=0 specifies that the data bytes following the header byte contain a raw byte sequence with byte code values defined in this specification. The VM loader only loads the byte code portion of the data bytes, not including the two header bytes.

0	0	C_0	C_1	…

4.6.5 pkEX Atom

The pkEX atom contains a list of Export entries, referred to in this document as an *Export Table*. The first 4 bytes encode a 32-bit unsigned integer in big-endian byte order equal to the number of entries that follow. Each following Export entry consists of a name, encoded as one byte containing the name size S followed by S bytes containing the ASCII characters of the name, including a terminating 0, followed by a 32-bit unsigned integer in big-endian byte order representing the byte offset of the named entry point. This offset is from the start of the byte code data stored in the pkCS atom.

Entry format:

N	C_0	C_1	…	C_{N-2}	0	O_0	O_1	O_2	O_3

Example:

5	‘M’	‘A’	‘I’	‘N’	0	0	0	0	64

Represents the entry point MAIN at offset 64.

4.6.6 pkRQ Atom

The pkRQ atom contains requirements that need to be met by the Virtual Machine implementation in order to execute this code. This atom is OPTIONAL. If no such atom is present in the code module, the VM will use a default implementation settings as MAY be defined by an implementation profile.

This atom consists of an array of 32-bit unsigned integer values, one for each requirement field:

Field Name	Description
vmVersion	Version ID of the VM Specification. Set to 0 for this specification.
minDataMemorySize	Minimum size in bytes of the data memory available to the code. This includes the data memory used to load the image of the Data Segment, as well as the data memory used by the Data Stack. The VM MUST refuse to load the module if it cannot satisfy this requirement.
minCallStackDepth	Minimum number of nested subroutine calls (`OP_JSR` and `OP_JSRR`) that must be supported by the VM. The VM MUST refuse to load the module if it cannot satisfy this requirement.
flags	Set of bit-flags to signal required features of the VM. A VM implementation MUST refuse to load a code module that has any unknown flag set. Since there are currently no flags defined, a VM implementation compliant with this specification MUST check that this flag is set to 0.

4.6.7 Module Loader

The Plankton VM is responsible for loading code modules.

When a code module is loaded, the Data Segment memory image encoded in the pkDS atom is loaded at a memory address in the Data Memory. That address is chosen by the VM Loader, and is stored in the DS pseudo-register when the code executes.

The Code Segment memory image encoded in the pkCS atom is loaded at a memory address in the Code Memory. That address is chosen by the VM Loader, and is stored in the CS pseudo-register when the code executes.

When a code module is loaded, the special routine named “Global.OnLoad” is executed if this routine is found in the entries of the Export Table (the list of Export entries contained in the pkEX atom of the code module, described in §4.6.5) . This routine takes no arguments on the stack, and returns an integer status upon return, 0 signifying success, and a negative error code signifying an error condition.

When a code module is unloaded (or when the VM that has loaded the module is disposed of), the special routine named “Global.OnUnload” is executed if it is found in the Export Table. This routine takes no arguments on the stack, and returns an integer status upon return, 0 signifying success, and a negative error code signifying an error condition.

4.7 *System Calls*

Plankton Programs can call functions implemented outside of their code module’s Code Segment. This is done through the use of the `OP_CALL` instruction, which takes an integer stack operand specifying the System Call Number to call. Depending on the system call, the code to be executed can be a Plankton byte code routine in a different code module (for instance, a library of utility functions), code executed directly by the VM in the VM’s native implementation format, or code in an external software module, such as the VM’s host environment.

If an `OP_CALL` instruction is executed with an operand that contains a number that does not correspond to any system call, the VM SHALL behave as if the `SYS_NOP` system call was called.

4.7.1 System Call Numbers Allocation

Plankton reserves System Call Numbers 0 to 1023 for fixed system calls, that is, system calls that have the same number on all VM implementations.

System Call Numbers 1024 to 16383 are available for the VM to assign dynamically. For example, the System Call Numbers returned by System.FindSystemCallByName can be allocated dynamically by the VM, and do not have to be the same numbers on all VM implementations.

There are currently several fixed System Call Numbers specified:

Mnemonic	Number	System Call
SYS_NOP	0	System.NoOperation
SYS_DEBUG_PRINT	1	System.DebugPrint
SYS_FIND_SYSTEM_CALL_BY_NAME	2	System.FindSystemCallByName
SYS_SYSTEM_HOST_GET_OBJECT	3	System.Host.GetObject
SYS_SYSTEM_HOST_SET_OBJECT	4	System.Host.SetObject

4.7.2 Standard System Calls

The current specification contains a few standard system calls that are useful for writing control programs. These calls include the fixed-number system calls listed in the table above, as well as system calls that have dynamically determined numbers (i.e., their System Call Number is retrieved by calling the System.FindSystemCallByName system call with their name passed as the argument).

All the system calls specified in this section that can return a negative error code MAY return error codes with any negative value. Section 4.7.4 defines specific values for which a description exists in this Plankton Virtual Machine specification. If negative error code values are returned that are not described in this specification, they MUST be interpreted as if they were the generic error code value FAILURE. Unless the specification for a system call gives a more specific description of certain error code values, the default descriptions from §4.7.4 apply.

4.7.2.1 System.NoOperation

Input: None

Output: None

Description: This call is a no-operation call. It just returns (does nothing else). It is used primarily for testing the VM.

4.7.2.2 System.DebugPrint

Input:

Top of stack:

Message
…

Message: Address of a memory location containing a null-terminated string.

Output: None

Description: Prints a string of text to a debug output. If the VM implementation does not include a facility to output debug text (such as would be the case in a non-development environment), the VM MAY ignore the call and do nothing except pop the message address from the top of the stack.

4.7.2.3 System.FindSystemCallByName

Input:

Top of stack:

Name
…

Name: Address of a null-terminated ASCII string containing the name of the system call to look for.

Output:

Top of stack:

Id
…

Id: System Call Number if the system call with that name is implemented, ERROR_NO_SUCH_ITEM if the system call is not implemented, or a negative error code if an error occurred.

Description: Find the number of a system call, given its name.

4.7.2.4 System.Host.GetLocalTime

Input: None

Output:

Top of stack:

LocalTime
…

LocalTime: The current value of the local time of the host.

Description: The local time is expressed as a 32-bit signed integer number equal to the number of minutes elapsed since January 1, 1970 00:00:00 UTC plus the difference in minutes between the local time and UTC, or a negative error code.

4.7.2.5 System.Host.GetLocalTimeOffset

Input: None

Output:

Top of stack

LocalTimeOffset
…

LocalTimeOffset: The current time offset (from UTC time) of the host.

Description: The time offset is expressed as a 32-bit signed integer number equal to the difference (in minutes) between local time and UTC time (i.e., LocalTime - UTC).

4.7.2.6 System.Host.GetTrustedTime

Input: None

Output:

Top of stack

TrustedTime
Flags
…

The robustness of the implementation of the following features is out of the scope of this specification.

TrustedTime: The current value of the trusted time clock, or a negative error code if the trusted time is not available. The value is expressed as a 32-bit signed integer number equal to the number of minutes elapsed since January 1, 1970 00:00:00 UTC, or a negative error code.

Flags: Bit-set of flags that further define the current state of the trusted clock. If an error has occurred (the value of TrustedTime is a negative error code), the value of Flags MUST be 0.

The following flags are defined:

Bit index (0 is LSB)	Name	Description
0	TIME_IS_ESTIMATE	The value of TrustedTime is known not to be at its most accurate value, and therefore SHOULD be considered an estimate.

Description: This system call is only relevant on systems that implement a trusted clock that can be synchronized with a trusted time source and maintain a monotonic time counter. The value of the trusted time is not guaranteed to always be accurate, but the following properties are REQUIRED to be true:

- The value of the trusted time is expressed as a UTC time value. (The trusted time is not in the local time zone, as the current locality usually cannot be securely determined.)
- The trusted time never goes back.
- The trusted clock does not advance faster than real time.

Therefore, the value of TrustedTime is always between the value of the last synchronized time (synchronized with a trusted time source) and the current real time. If the system is able to determine that its trusted clock has been operating and updating continuously and normally without interruption since the last synchronization with a trusted time source, it can determine that the value of TrustedTime is not an estimate, but an accurate value, and set the TIME_IS_ESTIMATE flag to 0.

If the trusted clock detects that a failure condition has occurred (hardware or software), and it is unable to return even an estimate of the trusted time, the implementation MUST return an error code, and the value of the returned Flags MUST be set to 0.

4.7.2.7 System.Host.GetObject

Input:

Top of stack:

Parent
Name
ReturnBuffer
ReturnBufferSize
…

Parent: 32-bit handle of the parent container.

Name: Address of a null-terminated string containing the path to the requested object, relative to the parent container.

ReturnBuffer: Address of a memory buffer where the value of the object is to be stored.

ReturnBufferSize: 32-bit integer indicating the size in bytes of the memory buffer where the value of the object is to be stored.

Output:

Top of stack:

TypeId
Size
…

TypeId: Object type ID, or negative error code if the call failed. If the requested object does not exist, the error returned is ERROR_NO_SUCH_ITEM. If the buffer supplied for the return value is too small, the error returned is ERROR_INSUFFICIENT_SPACE. If the part of the object tree that is being accessed is access-controlled, and the calling program does not have permission to access the object, ERROR_PERMISSION_DENIED is returned. Other error codes MAY be returned.

Size: 32-bit integer indicating the size in bytes of the data returned in the buffer supplied by the caller, or the size required if the caller provided a buffer that was too small.

Description: This system call is a generic interface that allows a program to access objects provided by the VM's Host.

4.7.2.7.1 Object Types

There are four types of host objects: Strings, Integers, Byte Arrays and Containers.

Object Type	Type ID Name	TypeId Value
Container	OBJECT_TYPE_CONTAINER	0
Integer	OBJECT_TYPE_INTEGER	1
String	OBJECT_TYPE_STRING	2
Byte Array	OBJECT_TYPE_BYTE_ARRAY	3

4.7.2.7.1.1 Byte Arrays

The value of a Byte Array object is an array of 8-bit bytes.

4.7.2.7.1.2 Strings

The value of a String object is a null-terminated character string encoded in UTF-8.

4.7.2.7.1.3 Integers

The value of an Integer object is a 32-bit signed integer value.

4.7.2.7.1.4 Containers

Containers are generic containers that contain a sequence of any number of objects of any combination of types. Objects contained in a container are called the children of that container.

The value of a container is a 32-bit container handle that is unique within a given VM instance.

The root container '/' has the fixed handle value 0.

4.7.2.7.2 Object Names

The namespace for host objects is a hierarchical namespace, where the name of a container's child object is constructed by appending the name of the child to the name of the parent container, separated by a '/' character. For example, if a container is named '/Node/Attributes', and has a String child named 'Type', then '/Node/Attributes/Type' refers to the child String.

The root of the namespace is '/'. All absolute names start with a '/'. Names that do not start with a '/' are relative names. Relative names are relative to a parent container. For example, the name 'Attributes/Type', relative to parent '/Node' is the object with the absolute name '/Node/Attributes/Type'.

4.7.2.7.2.1 Virtual Names and Virtual Objects

Some objects can be accessed by using virtual names. Virtual names are names that are not names attached to host objects, but a convention to identify either unnamed child objects, child objects with different names, or virtual child objects (child objects that are not real children of the container, but created dynamically when requested).

For any object, the following virtual names are defined:

	Virtual Name	Description
Virtual Name	@Name	Virtual string object: the name of the object. If the object is unnamed, the value is an empty string. Note that unnamed objects are only accessible through the @<n> virtual name of a container object (see below).
Virtual Size	@Size	Virtual integer object. The integer value is equal to the size in bytes required to store this object. For an integer, this is 4. For a string, it is the number of bytes needed to store the UTF-8 string plus a null byte terminator. For byte array, this is the number of bytes in the array.
Virtual Type	@Type	Virtual integer object. The integer value is equal to the object's TypeId.

For any container, the following virtual names are defined:

	Virtual Name	Description
Virtual Index	@<n>	The <n>th object in a container. The first object in a container has index 0. <n> is expressed as a decimal number. Example: if 'Attributes' is a container that contains 5 child objects, 'Attributes/@4' is the 5th child of the container.
Virtual Size	@Size	Virtual integer object. The integer value is equal to the number of objects in the container.

4.7.2.7.2.2 Examples

Let’s take an example of a hierarchy of host objects:

<table>
<tr><th>Name</th><th>Value</th><th colspan="5">Children</th></tr>
<tr><td rowspan="9">Node</td><td rowspan="9">1</td><th>Name</th><th>Value</th><th colspan="3">Children</th></tr>
<tr><td>Type</td><td>“Device”</td><td colspan="3"></td></tr>
<tr><th>Name</th><th>Value</th><th colspan="3">Children</th></tr>
<tr><td rowspan="6">Attributes</td><td rowspan="6">2</td><th>Name</th><th>Value</th><th>Children</th></tr>
<tr><td>Color</td><td>“Red”</td><td></td></tr>
<tr><th>Name</th><th>Value</th><th>Children</th></tr>
<tr><td>Size</td><td>78</td><td></td></tr>
<tr><th>Name</th><th>Value</th><th>Children</th></tr>
<tr><td>Domain</td><td>“TopLevel”</td><td></td></tr>
</table>

Calling System.Host.GetObject(parent=0, name=”Node”) returns the type ID 0 (Container), and writes the handle value 1 in the buffer supplied by the caller. (The size of the value is 4 bytes.)

Calling System.Host.GetObject(parent=0, name=”Node/Attributes/Domain”) returns the type ID 2 (String), and writes the string “TopLevel” in the buffer supplied by the caller. (The size of the value is 9 bytes.)

Calling System.Host.GetObject(parent=1, name=”Attributes/@1”) returns the type ID 1 (Integer), and writes the integer 78 in the buffer supplied by the caller. (The size of the value is 4 bytes.)

Calling System.Host.GetObject(parent=0, name=”DoesNotExist”) returns the error code ERROR_NO_SUCH_ITEM.

4.7.2.8 System.Host.SetObject

Input:

Top of stack:

Parent
Name
ObjectAddress
ObjectType
ObjectSize
…

Parent: 32-bit handle of the parent container.

Name: Address of a null-terminated string containing the path to the object, relative to the parent container.

ObjectAddress: Address of a memory buffer where the value of the object is stored. If the address is 0, the call is interpreted as a request to destroy the object. The data at the address depends on the type of the object.

ObjectType: The type ID of the object.

ObjectSize: 32-bit integer indicating the size in bytes of the memory buffer where the value of the object is stored. For Integer objects, the size MUST be set to 4. For String objects, the size is the size of the memory buffer, including the null terminator. For Byte Array objects, the size is the number of bytes in the array.

Output:

Top of stack:

ResultCode
…

ResultCode: 0 if the call succeeded, or negative error code if the call failed. If the call is a request to destroy an object and the requested object does not exist, or the call is a request to create or write an object and the object's parent does not exist, the error code returned is ERROR_NO_SUCH_ITEM. If the part of the object tree that is being accessed is access-controlled, and the calling program does not have permission to access the object, ERROR_PERMISSION_DENIED is returned. Other error codes MAY be returned.

Description: This system call is a generic interface that allows a program to create, write and destroy objects provided by the VM's host. The description of the object names and types is the same as for System.Host.GetObject.

Not all host objects support being written to or destroyed, and not all containers support having child objects created. When a SetObject call is made for an object that does not support the operation, ERROR_PERMISSION_DENIED is returned.

4.7.2.8.1 Creating a Container

There is a special case where the object refers to a container and the ObjectAddress is not 0. In this case the ObjectSize parameter MUST be set to 0. The value of ObjectAddress is ignored. If the container already exists, nothing is done, and a SUCCESS ResultCode is returned. If the container does not exist, and the parent of the container is writeable, an empty container is created.

4.7.2.8.2 Destroying an Object

If the ObjectAddress is 0, the call is interpreted as a request to destroy the object.

4.7.2.9 Octopus.Links.IsNodeReachable

Input:

Top of stack:

NodeId
…

__NodeId__: Address of a null-terminated string containing the ID of the target Link to be tested for reachability.

Output:

Top of stack:

ResultCode
StatusBlockAddress
…

__ResultCode__: An integer value. The result value is 0 if the Node is reachable, or a negative error code if it is not.

__StatusBlockAddress__: 0 SHALL be returned.

Description: This system call is used by control programs to check whether a given Node is reachable from the Node associated with the entity hosting this instance of the Plankton VM.

4.7.2.10 System.Host.SpawnVm

Input:

Top of stack:

ModuleId
IdentityRequirementsBlockAddress
…

__ModuleId__: Address of a null-terminated string containing the ID of the code module to be loaded into the VM. It is up to the specification of the VM's host to describe the mechanism by which the actual code module corresponding to this module ID is to be located.

__IdentityRequirementsBlockAddress__: Address of an IdentityRequirementsBlock as defined below. If this address is 0, no identity requirements are set.

Output:

Top of stack:

ResultCode
VmHandle
…

__ResultCode__: An integer value. The result value is 0 if the call was successful, or a negative error code if it failed.

__VmHandle__: An integer value specifying the handle to the instance of the VM that has been created. (If the call failed, this is set to 0.) That handle is only guaranteed to be unique within the VM in which this call is made.

Description: This system call is used by control programs to request that a new instance of a Plankton Virtual Machine be created, and a code module loaded.

The caller MAY specify identity and signature requirements for the code module that is to be loaded by providing an IdentityRequirementsBlock that contains a list of identity names that are acceptable, and that indicates whether content-key-based signatures are required. If the code module to be loaded does not meet the requirements, the system call fails with a ResultCode equal to ERROR_NO_SUCH_ITEM. The requirements are met if and only if the RequireContentKeySignature requirement of the IdentityRequirementsBlock is met and one or more of the identity names listed in the IdentityRequirementsBlock is included in the list of the identity names applicable to the code module to be loaded. When IdentityNameCount is 0, the list of identity names is empty and the identity requirement is met.

The host of the newly created VM exposes the same host objects as the ones exposed to the caller, with the exception that the host object "/Octopus/Runtime/Parent/Id" is set to the identity of the caller. This host object is a container. The children of this container are objects of type String, each with a value representing a name. The semantics and specific details of those names are specified by the specification of the VM's host.

When the VM that is running the code for the caller terminates, any spawned VM that has not been explicitly released by calling System.Host.ReleaseVm is automatically released by the system as if System.Host.ReleaseVm had been called.

IdentityRequirementsBlock:

Name	Type
RequireContentKeySignature	32-bit unsigned integer
IdentityNameCount	32-bit unsigned integer
IdentityNameAddress<1>	32-bit address
…	…
IdentityNameAddress<N>	32-bit address

RequireContentKeySignature: An integer value equal to 0 or 1. When the value is 1, the code module meets the requirement if and only if there is a set of valid direct or indirect signatures with the same content keys as the ones that are required to assert the validity of the caller. When the value is 0, content key signatures are not required.

IdentityNameCount: The number of 32-bit addresses following this field.

IdentityNameAddress<n>: Address of a null-terminated UTF-8 encoded string specifying an acceptable identity name.

4.7.2.11 System.Host.CallVm

Input:

Top of stack:

VmHandle
EntryPoint
ParameterBlockAddress
ParameterBlockSize

ReturnBufferAddress
ReturnBufferSize
…

VmHandle: An integer value specifying the handle of a VM that was created by calling System.Host.SpawnVm.

EntryPoint: Address of a null-terminated string that specifies the name of the entry point to call. This name needs to match one of the entry points in the Export Table of the code module that was loaded into the VM instance that corresponds to the VmHandle parameter.

ParameterBlockAddress: Address of a memory block that contains data to be passed to the callee. If no parameters are passed to the callee, this is set to 0.

ParameterBlockSize: Size in bytes of the memory block at address ParameterBlockAddress, or 0 if ParameterBlockAddress is 0.

ReturnBufferAddress: Address of a memory buffer where the caller can receive data from the callee. If the caller does not expect any data back from the callee, this is set to 0.

ReturnBufferSize: Size in bytes of the memory buffer at address ReturnBufferAddress, or 0 if ReturnBufferAddress is 0.

Output:

Top of stack:

SystemResultCode
CalleeResultCode
ReturnBlockSize
…

SystemResultCode: An integer value. The result value is 0 if the call was successful, or a negative error code if it failed. This value is determined by the system, not by the callee. Success only indicates that the system was able to successfully find the routine to call, execute the routine and get the return value from the routine. The return value from the routine itself is returned in the CalleeResultCode value.

CalleeResultCode: An integer value. This is the value returned by the callee.

ReturnBlockSize: The size in bytes of the data returned in the buffer supplied by the caller, or the size required if the caller provided a buffer that was too small. If no data was returned by the callee, the value is 0.

Description: This system call is used by control programs to call routines that are implemented in code modules loaded in VM instances created using the System.Host.SpawnVm system call.

The called routine MUST comply with the following interface conventions:

When the routine is called, the stack contains the value ParameterBlockSize supplied by the caller, indicating the size of the parameter block at the top, followed by ParameterBlockSize bytes of data. If the size is not a multiple of 4, the data on the stack will be padded with 0-value bytes to ensure that the Stack Pointer remains a multiple of 4.

Top of stack:

ParameterBlockSize
…data…

Upon return, the routine MUST provide the following return values on the stack:

Top of stack:

ResultCode
ReturnBlockAddress
ReturnBlockSize
…

ResultCode: An integer value. The result value is 0 if the call was successful, or a negative error code if it failed.

ReturnBlockAddress: Address of a memory block that contains data to be returned to the caller. If no data is returned, this is set to 0.

ReturnBlockSize: Size in bytes of the memory block at address ReturnBlockAddress, or 0 if ReturnBlockAddress is 0.

4.7.2.12 System.Host.ReleaseVm

Input:

Top of stack:

VmHandle
…

VmHandle: An integer value. Handle of a VM that was created by calling System.Host.SpawnVm.

Output:

Top of stack:

ResultCode
…

ResultCode: An integer value. The result value is 0 if the call was successful or a negative error code if it failed.

Description: This system call is used by control programs to release a VM that was spawned by a previous call to System.Host.SpawnVm. Any VMs spawned by the released VM are released, and so on, recursively.

4.7.3 Standard Data Structures

The following are standard data structures used by some of the standard system calls. They may also be used by any custom system call, as functions of the VM's host.

4.7.3.1 Standard Parameters

4.7.3.1.1 Parameter Block

ParameterBlock:

Name	Type
Name	NameBlock
Value	ValueBlock

Name: Name of the parameter

Value: Value of the parameter

4.7.3.1.2 Extended Parameter Block

ExtendedParameterBlock:

Name	Type
Flags	32-bit bit field
Parameter	ParameterBlock

Flags: Vector of Boolean flags

Parameter: Parameter block containing a name and a value

4.7.3.1.3 Name Block

NameBlock:

Name	Type
Size	32-bit integer
Characters	Array of 8-bit characters

Size: 32-bit unsigned integer equal to the size in bytes of the Characters field that follows. If this value is 0, the Characters field is left empty (nothing follows).

Characters: NULL-terminated UTF-8 string

4.7.3.1.4 Value Block

ValueBlock:

Name	Type
Type	32-bit integer
Size	32-bit integer
Data	Array of 8-bit bytes

Type: 32-bit type identifier. The following types are defined:

Type Identifier	Type Name	Description
0	Integer	32-bit integer value, encoded as 4 8-bit bytes in big-endian byte order. Unless otherwise specified, the value is considered signed.
1	Real	32-bit floating point value, encoded as IEEE-754 in big-endian byte order.
2	String	Null-terminated UTF-8 string.
3	Date	32-bit unsigned integer value, representing the number of minutes elapsed since January 1, 1970 00:00:00. Unless otherwise specified, the value is considered to be a UTC date. The most significant bit MUST be 0.
4	Parameter	ParameterBlock structure.
5	ExtendedParameter	ExtendedParameterBlock structure.
6	Resource	The value is a resource. The resource here is referenced by ID: the Data field of the value is a null-terminated ASCII string containing the ID of the resource that should be de-referenced to produce the actual data.
7	ValueList	An array of values (encoded as a ValueListBlock).
8	ByteArray	The value is an array of 8-bit bytes.

Size: 32-bit unsigned integer equal to the size in bytes of the Data field that follows. If this value is 0, the Data field is left empty (nothing follows).

Data: Array of 8-bit bytes representing a value. The actual bytes depend on the data encoding specified by the Type field.

4.7.3.1.5 Value List Block

ValueListBlock:

Name	Type
ValueCount	32-bit integer
Value0	ValueBlock
Value1	ValueBlock
…	…

ValueCount: 32-bit unsigned integer equal to the number of ValueBlock structures that follow. If this value is 0, no ValueBlock follows.

Value0, Value1, …: Sequence of 0 or more ValueBlock structures.

4.7.3.2 Standard Extended Status Block

The standard ExtendedStatusBlock is a data structure typically used to convey extended information as a return status from a call to a routine or a system call. It is a generic data structure that can be used in a variety of contexts, with a range of different possible values for its fields.

ExtendedStatusBlock:

Name	Type
GlobalFlags	32-bit bit field
Category	32-bit integer
SubCategory	32-bit integer
LocalFlags	32-bit bit field
CacheDuration	CacheDurationBlock
Parameters	ValueListBlock

GlobalFlags: Boolean flags whose semantics are the same regardless of the Category field. The position and meaning of the flags are defined by profiles that use standard ExtendedStatusBlock data structures.

Category: Integer unique identifier of a category to which this status belongs. The category identifier values are defined by profiles that use standard ExtendedStatusBlock data structures.

SubCategory: Integer identifier (unique within the category) of a subcategory that further classifies the type of status described by this block.

LocalFlags: Boolean flags whose semantics are local to the category and subcategory of this status block. The position and meaning of the flags are defined by profiles that define and use the semantics of the category.

CacheDuration: Indicates the duration for which this status can be cached (i.e., how long it remains valid). See the CacheDurationBlock definition below for the actual value of the duration.

Parameters: List of 0 or more ValueBlocks. Each ValueBlock contains a parameter encoded as a value of type Parameter or ExtendedParameter. Each parameter binds a name to a typed value, and is used to encode flexible variable data that describes the status block in more detail than just the category, subcategory, cache duration and flags.

CacheDurationBlock:

Name	Type
Type	32-bit integer
Value	32-bit integer

Type: Integer identifier for the type of the value. The following types can be used:

0	The value is a 32-bit unsigned integer that represents the number of seconds from the current time that the status can be cached. (A value of 0 means that the status cannot be cached at all, and therefore can only be used once.) The special value 0xFFFFFFFF is interpreted as an infinite duration; the status can be cached indefinitely.
1	The value is a 32-bit unsigned integer that represents an absolute local time, expressed as the number of minutes elapsed since January 1, 1970 00:00:00. The most significant bit MUST be 0.

Value: 32-bit integer. The meaning of this value depends on the Type field.

4.7.4 Standard Result Codes

Standard result codes are used in various APIs. Other result codes may be defined for use in more specific APIs.

Integer Value	Name	Description
0	SUCCESS	Success
-1	FAILURE	Unspecified failure
-2	ERROR_INTERNAL	An internal (implementation) error has occurred
-3	ERROR_INVALID_PARAMETER	A parameter has an invalid value
-4	ERROR_OUT_OF_MEMORY	Not enough memory available to complete successfully
-5	ERROR_OUT_OF_RESOURCES	Not enough resources available to complete successfully
-6	ERROR_NO_SUCH_ITEM	The requested item does not exist or was not found
-7	ERROR_INSUFFICIENT_SPACE	Not enough memory space supplied by the caller (typically used when a return buffer is too small)
-8	ERROR_PERMISSION_DENIED	The permission to perform the call is denied to the caller
-9	ERROR_RUNTIME_EXCEPTION	An error has occurred during the execution of byte code
-10	ERROR_INVALID_FORMAT	Error caused by data with an invalid format (for example, invalid data in a code module)

5 Octopus Object Serialization

5.1 *Introduction*

The Octopus Object Serialization specification provided in this section of the document defines an encoding-neutral way of computing a canonical byte sequence (CBS) for Octopus objects. The purpose of this canonical byte sequence is the computation of digests for the digital signature of objects. This byte sequence is independent of the way the objects are represented or transmitted.

5.2 *Canonical Byte Sequence Algorithm*

The canonical byte sequence algorithm consists of constructing sequences of bytes from values of fields. Each field has a value with a simple type or a compound type. Some fields can be specified to be optional (the field may be present or omitted).

5.2.1 Simple Types

Simple types are:

- Integer
- String
- Byte
- Boolean

5.2.2 Compound Types

Compound types consist of one or more subfields. Each subfield has a value with a simple or compound type.

Compound types are either heterogeneous or homogenous:

- Heterogeneous: one or more subfield values of different types (simple or compound)
- Homogeneous: one or more subfield values of the same type (simple or compound)

5.2.3 Encoding Rules

The canonical byte sequence of a field that is always present is obtained by applying the appropriate encoding rule (depending on the type of the field) to the field's value. The canonical byte sequence of a field that is specified to be optional is obtained by applying the encoding rule for optional fields, as defined below. In the following encoding rule descriptions, the term *byte* means an 8-bit value (octet):

Optional Fields

0

or

1	Field

If an optional field is present, its value is serialized as the byte value 1 followed by the canonical byte sequence of the field value. If it is omitted, its value is serialized as the byte value 0.

Heterogeneous Compound

Field 0	Field 1	Field 2	…

The canonical byte sequence for a heterogeneous compound is the concatenation of the canonical byte sequences of each subfield value. Optional fields are not skipped, but serialized according to the rule for optional fields.

Homogeneous Compound

Field count	Field 0	Field 1	…

The canonical byte sequence for a homogeneous compound is the subfield count, encoded as a sequence of 4 bytes in big-endian order, followed by the concatenation of each subfield value's canonical byte sequence. If the subfield count is 0, then nothing follows the 4-byte field count; in this case, all 4 bytes have the value 0.

Integer

I_0	I_1	I_2	I_3

The canonical byte sequence for an integer field is the 32-bit integer value, encoded as a sequence of 4 bytes, in big-endian order.

String

Byte Count	Byte 0	Byte 1	…

The canonical byte sequence for a string field is a byte count followed by the UTF-8 encoded byte sequence for the string (not null-terminated).

The Byte Count of the encoded byte sequence is encoded as a sequence of 4 bytes in big-endian order.

The Byte Count is followed by the sequence of bytes of the UTF-8 encoded string.

Byte

B

The canonical byte sequence for a byte field is its 8-bit value.

Boolean

B

The canonical byte sequence for a Boolean field is an 8-bit value: 0 for false, and 1 for true.

5.3 *Application to Octopus Objects*

The canonical byte sequence for an Octopus object is the concatenation of the canonical byte sequences of each of its fields, in the order they are defined in the object model (see §2).

For heterogeneous compound types, the order of the fields is the one specified in the type definition. For homogeneous compound types, the order of the elements is specified in the following sections 5.3.1 through 5.3.4.

5.3.1 Attributes

An Octopus object's 'attributes' field, as well as the 'attributes' field of attributes of type ListAttribute, is encoded as a homogeneous compound in which the fields are all of type Attribute (§2.4.1.1) , and are sorted lexicographically by their 'name' field. Attributes contained in the 'attributes' field of attributes of type ArrayAttribute are not sorted; they are serialized in their array order.

5.3.2 Extensions

An Octopus object's internal extensions (§2.4.1.2) are sorted lexicographically by their 'id' field.

For internal extensions, the 'ExtensionData' field is NOT used in the computation of the canonical byte sequence. For such extensions, if they need to be included in the computation of a digest for the purpose of a signature, they will contain a 'digest' field that will represent the digest of the actual data carried in the 'ExtensionData'. For each type of extension data, a definition will be given that allows the computation of its canonical byte sequence.

5.3.3 Controller

ContentKey references are sorted lexicographically by their 'id' field.

5.3.4 ScubaKeys

The keys in the 'publicKeys', 'privateKeys' and 'secretKeys' fields (see §6.6.1) are sorted lexicographically by their 'id' field.

5.4 *Example*

```
class X {
    int i;
    int j;
}

class A {
    int a[];
    string s;
}
```

```
class B extends A {
    {X optional_x;}
    X x;
    (string toDiscardInCano;)
    string s2;
}
```

The canonical byte sequence of an instance of class B where a[] = {7,8,9}, s = “Abc”, x = {5,4}, s2=””, and optional_x is not present is serialized as:

3	7	8	9	3	“Abc” as UTF-8	0	Cano(X)	0
4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	3 bytes	1 byte	8 bytes	4 bytes

where Cano(X) is:

5	4
4 bytes	4 bytes

6 Scuba Key Distribution

6.1 *Introduction*

This section of the document provides the specification for Scuba, a key distribution system that has been designed to fit very naturally within the Octopus architecture.

The basic principle behind Scuba is to use the Octopus Link objects (§2.3.2) to distribute keys, in addition to their primary purpose of establishing relationships between Node objects. An Octopus Control object contains a control program that decides whether or not a requested action should be granted. That control program often checks whether a specific Octopus Node is reachable via a chain of Octopus Links. Scuba makes it possible to take advantage of the existence of that chain of Links to facilitate the distribution of a key such that it is available to the Octopus Engine that is executing the control.

Each Octopus Node object (§2.3.1) used in an Octopus deployment that uses the Scuba Key Distribution system has Scuba keys. Those keys are used to encrypt content keys and other Nodes' Scuba keys. Each Octopus Link object created for use in the same deployment contains some cryptographic Scuba data payload that allows key information to be derived when chains of Links are processed by the Octopus Engine.

With Nodes and Links carrying Scuba keys this way, given a chain of Links from a Node A to a Node Z, any entity (such as the Octopus Engine of a client host application) that has access to the private Scuba sharing keys of A also has access to the private Scuba sharing keys of Z. Having access to Z's private Scuba sharing keys gives the entity access to any content key encrypted with those keys.

Each Octopus deployment will select a public key cipher (such as RSA) and a symmetric secret key cipher (such as AES) to use for the implementation of this key distribution system.

6.2 *Nodes, Entities and Scuba Keys*

6.2.1 Entities

In an Octopus system, Nodes are data objects, not active participants. Active participants, in this context, are called entities. Examples of entities are media players, devices, content packagers, etc. Entities typically have Octopus Nodes associated with them. An entity that consumes content uses an Octopus Engine and manages at least one Node object that constitutes its Octopus Personality. An entity is assumed to have access to all the data of the Node objects it manages, including all the private information of those objects.

6.2.2 Nodes

Node objects that participate in a Scuba key distribution system contain Scuba keys as part of their data. There are two categories of Scuba keys: *sharing keys* and *confidentiality keys*. In each category there are three types of keys: public keys, private keys, and secret symmetric keys. The following sections define the different key categories and types that Scuba can use.

Note that a specific deployment of the technology may use a subset of these keys. For example, a system could be configured to only work with public and private keys, omitting the use of secret symmetric keys. Or a system could be deployed without provisioning Nodes with confidentiality keys if it only makes use of sharing keys.

Public keys MUST be carried as internal extensions of the Node object. Private and symmetric secret keys MUST NOT be carried as internal extensions of Node objects.

6.2.2.1 Sharing Keys

Sharing keys are key pairs and/or symmetric keys that are shared by a Node N and all the Nodes Px for which there exists a Link from Px to N that contains Scuba key derivation extensions. We refer to this as downstream sharing.

- Scuba Sharing Public Key

  Kpub-share[N]

  This is the public part of a pair of public/private keys for the public key cipher. This key typically comes with a certificate so that its credentials can be verified by entities that want to cryptographically bind confidential information to it.

- Scuba Sharing Private Key

  Kpriv-share[N]

  This is the private part of the public/private key pair. The entity that manages Node N is responsible for ensuring that this private key is kept secret. For that reason, this private key will be stored and transported separately from the rest of the Node information. This private key will be shared downstream with other Nodes through the key derivation extensions of Links.

- Scuba Sharing Symmetric Key

  Ks-share[N]

  This is a key to be used with the symmetric cipher. Like the private key, this key is confidential, so the entity that manages Node N is responsible for keeping this key secret. This secret key will be shared downstream with other Nodes through the key derivation extensions of Links.

6.2.2.2 Confidentiality Keys

Confidentiality keys are key pairs and/or symmetric keys that are only known to the entity that manages the Node to which they belong. The difference between those keys and the sharing keys described above is that confidentiality keys will not be shared with other Nodes through the Scuba key derivation extensions in Links.

- Scuba Confidentiality Public Key

  Kpub-conf[N]

  This is the public part of a pair of public/private keys for the public key cipher. This key typically comes with a certificate so that its credentials can be verified by entities that want to cryptographically bind confidential information to it.

- Scuba Confidentiality Private Key

Kpriv-conf[N]

This is the private part of the public/private key pair. The entity that manages Node N is responsible for ensuring that this private key is kept secret. For that reason, this private key will be stored and transported separately from the rest of the Node information.

- Scuba Confidentiality Symmetric Key

Ks-conf[N]

This is a key to be used with the symmetric cipher. Like the private key, this key is confidential, so the entity that manages Node N is responsible for keeping this key secret.

6.3 *Cryptographic Elements*

Scuba can be implemented using different cryptographic algorithms. Scuba is not restricted to any specific choice of cryptographic algorithm. Nevertheless, it is necessary that, for a given deployment or profile, all participating entities agree on a set of supported algorithms. Any deployment will include support for at least one public key cipher (such as RSA) and/or one symmetric key cipher (such as AES).

For the purpose of this document, we will use the following notations for referring to cryptographic functions:

Ep(Kpub[N], M) means “the message M encrypted with the public key Kpub of Node N, using a public key cipher”

Dp(Kpriv[N], M) means “the message M decrypted with the private key Kpriv of Node N, using a public key cipher”

Es(Ks[N], M) means “the message M encrypted with the symmetric key Ks of Node N, using a symmetric key cipher”

Ds(Ks[N], M) means “the message M decrypted with the symmetric key Ks of Node N, using a symmetric key cipher”

6.4 *Binding of Content Keys*

There are two types of cryptographic binding used in Scuba. Binding a content key to a target Octopus Node’s sharing keys means making that key available to all entities that share the private Scuba sharing keys of that target Node. Binding a content key to a Node’s confidentiality keys means making that key available only to the entity that manages that Node. Binding is done by encrypting the key CK carried in a ContentKey object using one or both of the following methods:

- Public Binding

Create a ContentKey object that contains *Ep(Kpub[N], CK)*

- Symmetric Binding

Create a ContentKey object that contains *Es(Ks[N], CK)*

Whenever possible, Symmetric Binding SHOULD be used, as it uses a less computationally intensive algorithm, and therefore makes it less onerous for the receiving entity. However, the entity that creates the ContentKey object does not always have access to Ks[N]. In that case, the Public Binding MUST be used. Kpub[N] is not confidential and therefore can easily be made available to entities that need to perform the Public Binding. Kpub[N] will usually be made available to entities that need to bind content keys, accompanied by a certificate that can be inspected by the entity to decide whether Kpub[N] is indeed the key of a Node that can be trusted to handle the content key in accordance with some agreed-upon policy.

6.5 *Derivation of Scuba Keys using Links*

To allow an entity to have access to the Scuba sharing keys of all the Nodes reachable from its Personality Node, Link objects contain a Scuba extension payload. That payload allows any entity that has access to the private/secret Scuba keys of the Link's 'from' Node to also have access to the private/secret Scuba keys of the Link's 'to' Node. This way, an entity can decrypt any content key bound to a Node that is reachable from its Personality Node (if the binding was done using the target Node's sharing keys).

When an Octopus Engine processes Link objects, it processes the Scuba payload of each Link in order to update an internal chain of Scuba keys to which it has access.

The Scuba extension payload of a Link L from Node F to Node T SHALL consist of either:

- Public Derivation information

  Ep(Kpub-share[F], {Ks-share[T],Kpriv-share[T]})

or

- Symmetric Derivation information

  Es(Ks-share[F], {Ks-share[T],Kpriv-share[T]})

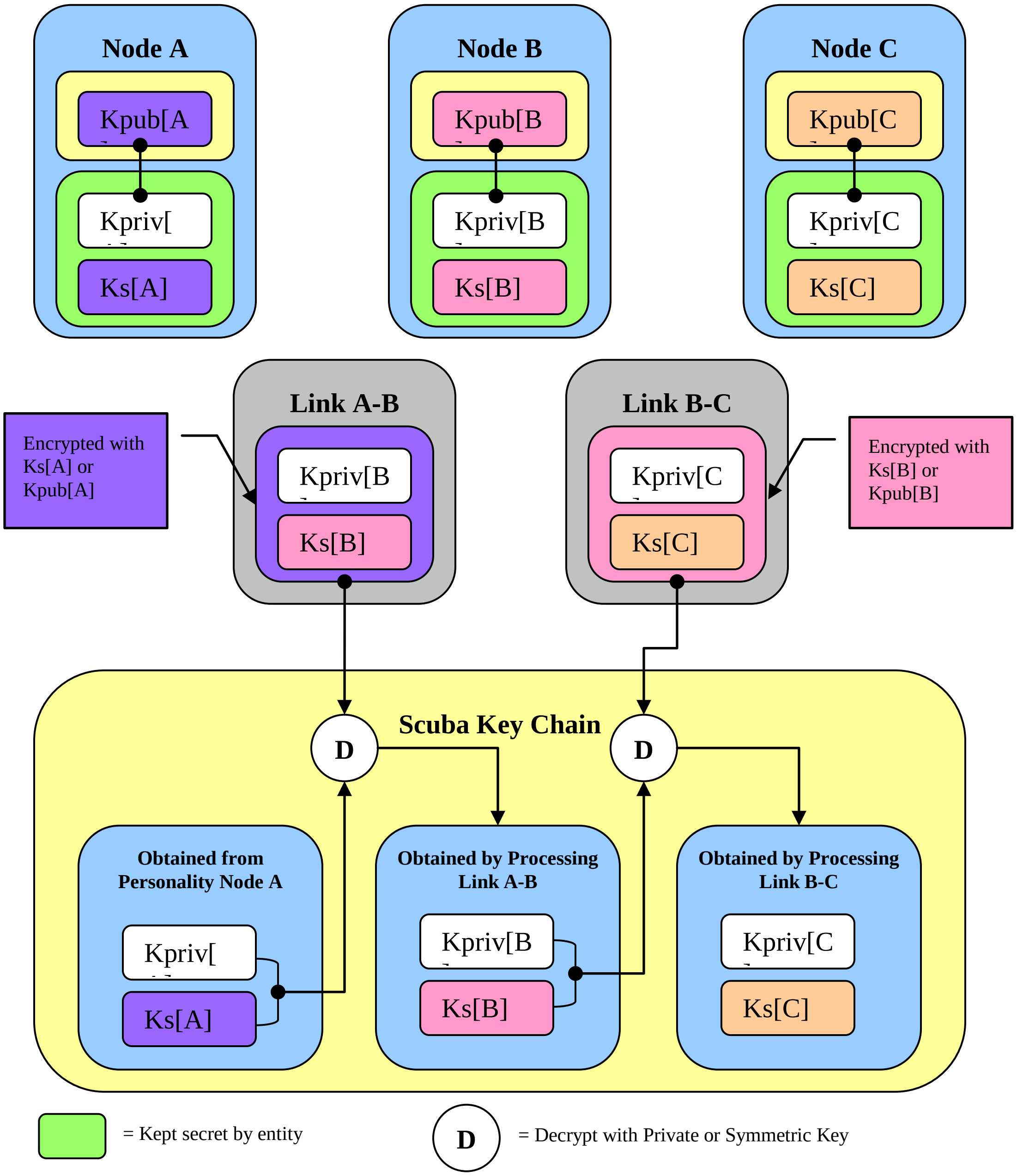
where {Ks-share[T],Kpriv-share[T]} is a data structure containing Ks-share[T] and Kpriv-share[T].

The Public Derivation information is used to convey the private Scuba sharing keys of Node T, Ks-share[T] and Kpriv-share[T], to any entity that has access to the private Scuba key of Node F, Kpriv-share[F].

The Symmetric Derivation information is used to convey the private Scuba sharing keys of Node T, Ks-share[T] and Kpriv-share[T], to any entity that has access to the symmetric Scuba sharing key of Node F, Ks-share[F].

Just as for binding content keys to Nodes, the preferred payload to include in a Link is the Symmetric Derivation information. This is only possible when the Link creator has access to Ks-share[F]. If not, then the Link creator will fall back to including the Public Derivation information as the Scuba payload for the Link.

Assuming that the Octopus Engine processing this Link already had Ks-share[F] and Kpriv-share[F] in its internal Scuba key chain, after processing the Link L[F->T] it will also have Ks-share[T] and Kpriv-share[T] on the key chain.


Node A
Kpub[A
Kpriv[
Ks[A]
Node B
Kpub[B
Kpriv[B
Ks[B]
Node C
Kpub[C
Kpriv[C
Ks[C]
Encrypted with Ks[A] or Kpub[A]
Link A-B
Kpriv[B
Ks[B]
Link B-C
Kpriv[C
Ks[C]
Encrypted with Ks[B] or Kpub[B]
D
Scuba Key Chain
D
Obtained from Personality Node A
Kpriv[
Ks[A]
Obtained by Processing Link A-B
Kpriv[B
Ks[B]
Obtained by Processing Link B-C
Kpriv[C
Ks[C]
= Kept secret by entity
D
= Decrypt with Private or Symmetric Key

6.6 *Data Structures*

6.6.1 ScubaKeys

```
class ScubaKeys {
    PairedKey [] publicKeys;
    PairedKey [] privateKeys;
    Key[]        secretKeys;
}

class ScubaKeysExtension extends ExtensionData(type='ScubaKeys') {
    ScubaKeys scubaKeys;
}
```

The Key and PairedKey data structures are defined in §2.4.5, and the ExtensionData data structure is defined in §2.4.1.2.

6.7 *Signatures and Flags*

ScubaKeys can appear in different contexts:

- Nodes: ScubaKeys is carried in an internal extension that MUST be signed as a part of the Node object. In this internal extension, the ScubaKeys MUST only contain the Node's public keys. The 'critical' flag of that extension SHOULD be set to 'false'.
- Links: ScubaKeys is carried in an internal extension and contains the private and/or symmetric sharing keys of the "to" Node. In this context, the ScubaKeys extension MUST NOT be signed. The 'critical' flag of that extension SHOULD be set to 'false'.

7 SeaShell Object Store

7.1 *Introduction*

This section of the document contains the SeaShell specification. SeaShell is a secure Object Store that can be used by Octopus Engine implementations to provide a secure state storage mechanism. Such a facility is useful to enable control programs to be able to read and write in a protected state database that is persistent from invocation to invocation. Such a state database can be used to store state objects such as play-counts, date of first use, accumulated rendering times, etc.

7.2 *Database Objects*

A SeaShell database contains objects. Objects are arranged in a logical hierarchy, where container objects are parents of their contained children objects. There are four types of objects. Each object has associated metadata and a type. Depending on its type, an object can also have a value.

SeaShell objects can be accessed from Plankton programs using the System.Host.GetObject and System.Host.SetObject system calls. The object metadata is accessed using Virtual Names. (See §7.4 for more details on object access.) Some of the metadata fields can be changed by clients of the SeaShell database, but some metadata fields are read-only (RO).

7.2.1 Object Metadata

Field	Type	Accessibility	Description
Name	String	RO	Name of the object. Only the following characters are allowed as object names (all others are reserved): a-z, A-Z, 0-9, ‘_’, ‘-‘, ‘+’, ‘:’, ‘.’, ‘$’, ‘!’, ‘*’, ‘ ‘
Owner	String	RW	ID of the owner of the object.
CreationDate	Unsigned 32-bit integer	RO	Date at which the object was created, expressed as the number of minutes elapsed since Jan 1 1970 00:00:00 UTC.
ModificationDate	Unsigned 32-bit integer	RO	Date at which the object was last modified, expressed as the number of minutes elapsed since Jan 1 1970 00:00:00 UTC. For container objects, this is the date at which a child was last added to or removed from the container. For other objects, this is the date at which their value was last changed.

ExpirationDate	Unsigned 32-bit integer	RW	Date at which the object expires, expressed as the number of minutes elapsed since Jan 1 1970 00:00:00 UTC. If the value is 0, it means that the object does not expire.
Flags	32-bit bit field	RW	Set of Boolean flags indicating Boolean properties of the object.

7.2.1.1 Flags

The following metadata flags are defined

Bit index	**Name**	**Meaning**
0 (LSB)	PUBLIC_READ	If set, indicates that the access control for this object is such that any client can read the object and its metadata, regardless of its identity.

7.2.2 Object Types

7.2.2.1 String

The value of a String object is a UTF-8 encoded character string.

7.2.2.2 Integer

The value of an Integer object is a 32-bit integer value.

7.2.2.3 Byte Array

The value of a Byte Array object is an array of bytes.

7.2.2.4 Container

A container object contains zero or more objects.

A container object is referred to as the parent of the objects it contains. The contained objects are referred to as the children of the container. All the container objects that make up the chain of an object’s parent, the parent’s parent, and so on, are called the object’s ancestors. If an object has another object as it ancestor, that object is called a descendant of the ancestor object.

7.3 Object Lifetime

The lifetimes of objects in a SeaShell database follow a number of rules. Objects can be explicitly destroyed, or implicitly destroyed. Objects can also be destroyed as the result of a database garbage collection.

Regardless of how an object is destroyed, the following rules MUST apply:

- The ModificationDate for the parent container of that object is set to the current time.

- If the object is a container, all its children are destroyed.

7.3.1 Explicit Object Destruction

Explicit object destruction happens when a client of the database requests that an object be removed (see §7.4 for more details on how this can be done using the Host.SetObject Plankton system call).

7.3.2 Implicit Object Destruction

Implicit object destruction happens when an object is destroyed as the result one of the objects in its ancestry being destroyed.

7.3.3 Garbage Collection

A SeaShell object database destroys any object that has expired. An object is considered to have expired when the time is later than the ExpirationDate field of the object's metadata. An implementation MAY periodically scan the database for expired objects and destroy them, or it MAY wait until an object is accessed to check its expiration date. An implementation MUST NOT return to a client an expired object.

When a container object is destroyed because it has expired, its children objects are also destroyed (and all their descendants, recursively) even if they have not expired yet.

7.4 *Object Access*

The objects in a SeaShell database can be accessed from Plankton programs through a pair of system calls: System.Host.GetObject to read the value of an object, and System.Host.SetObject to create, destroy or set the value of an object.

To be visible as a tree of host objects, a SeaShell database needs to be "mounted" under a certain name in the host object tree. This way, a database is visible as a sub-tree in the more general tree of host objects. To achieve this, all SeaShell databases contain a top-level built-in root container object that always exists. This root container is essentially the name of the database. All other objects in the database will be descendants of the root container. Multiple SeaShell databases can be mounted at different places in the host object tree. (For two databases to be mounted under the same host container, they need to have different names for their root containers.) For example, if a SeaShell database, whose root container is named 'Database1', contains a single Integer child object named 'Child1', the database could be mounted under the host object container "/SeaShell", in which case the 'Child1' object would be visible as '/SeaShell/Database1/Child1'.

All accesses to objects are governed by an access policy. See §7.5 on access control for more details.

7.4.1 Reading Objects

The value of an object can be read by using the system call System.Host.GetObject (§4.7.2.7). The four object types (Integer, String, Byte Array and Container) that can exist in the database map directly onto their counterparts in the Plankton Virtual Marchine specification (§4.7.2.7). The object values are accessed in the normal way, and the standard virtual names must be implemented. (Virtual names are described in the Plankton specification in §4.7.2.7.2.)

7.4.2 Creating Objects

Objects can be created by calling System.Host.SetObject (§4.7.2.8) for an object name that does not already exist. The object creation is done according to the system call specification. When an object is created, the SeaShell database

- sets the Owner field of the object metadata to the value of the Owner field of the parent container object's metadata;
- sets the CreationDate field of the metadata to the current time;
- sets the ModificationDate field of the metadata to the current time;
- sets the ExpirationDate field of the metadata to 0 (does not expire);
- sets the Flags field of the metadata to 0;
- sets the ModificationDate of the parent container to the current time.

As specified in the System.Host.SetObject system call specification, when creating an object under a path deeper than the existing container hierarchy, the SeaShell database will need to implicitly create the container objects that need to exist to create a path to the object being created. Implicit container object creation follows the same rules as an explicit creation.

For example, if there is a container "A" with no children, a request to set "A/B/C/SomeObject" will implicitly create containers "A/B" and "A/B/C" before creating "A/B/C/SomeObject".

7.4.3 Writing Objects

The value of objects can be changed by calling System.Host.SetObject for an object that already exists. If the specified ObjectType does not match the type ID of the existing object, ERROR_INVALID_PARAMETER is returned. If the type ID is OBJECT_TYPE_CONTAINER, no value needs to be specified (the ObjectAddress MUST be non-zero, but its value will be ignored).

When an existing object is set, the SeaShell database sets the ModificationDate of the object to the current time.

7.4.4 Destroying Objects

Objects can be explicitly destroyed by calling System.Host.SetObject for an object that already exists, with an ObjectAddress value of 0. (See the Plankton specification for the system call in §4.7.2.8.)

When an object is destroyed, the SeaShell database

- sets the ModificationDate of the parent container to the current time;
- destroys all its child objects if the destroyed object is a container.

7.4.5 Object Metadata

The metadata for SeaShell objects is accessed by using the System.Host.GetObject and System.Host.SetObject system calls with virtual names.

This table lists the standard and extended virtual names that are available for objects in a SeaShell database and how they map to the metadata fields, which are documented in §7.2.1.

Virtual Name	Type	Description
@Name	String	The Name field of the object metada
@Owner	String	The Owner field of the object metadata
@CreationDate	32-bit unsigned integer	The CreationDate field of the object metadata
@ModificationDate	32-bit unsigned integer	The ModificationDate field of the object metadata
@ExpirationDate	32-bit unsigned integer	The ExpirationDate field of the object metadata
@Flags	32-bit bit field	The Flags field of the object metadata

Metadata fields that are read-only (see the Metadata field description in §7.2) cannot be written to.

An implementation MUST refuse a request to set the Flags metadata field if one or more undefined flags (flags not defined in this specification) are set to 1. In this case, the return value for the System.Host.SetObject is ERROR_INVALID_PARAMETER. When reading the Flags metadata field, a client MUST ignore any flag not defined in this specification. When setting the Flags field of an object, a client MUST first read its existing value and preserve the value of any flag not defined in this specification.

7.5 *Object Ownership and Access Control*

Whenever a request to read, write, create, or destroy an object is made, the SeaShell database implementation first checks whether the caller has the permission to perform the request. The policy that governs access to objects is based on the concepts of principal identities and delegation. In order for the policy to be implemented, it is necessary that the trust model under which the implementation operates supports the notion of authenticated control programs. This is typically done by having the Plankton code module that contains the program be digitally signed (directly or indirectly through a secure reference) with the private key of a PKI key pair, and having a name certificate that associates a principal name with the signing key; however, different ways of determining control program identities are possible.

The access policy for the objects in a SeaShell database is comprised of a few simple rules:

- Read access to an object’s value is granted if the caller’s identity is the same as the owner of the object or if the PUBLIC_READ flag is set in the object’s Flags metadata field.

- Read access to an object’s value is granted if the caller has Read access to the object’s parent container.

- Write access to an object’s value is granted if the caller’s identity is the same as the owner of the object.

- Write access to an object’s value is granted if the caller has Write access to the object’s parent container.

- Create or Destroy access to an object is granted if the caller has Write access to the parent container of the object

- Read and Write access to an object’s metadata (using virtual names) follows the same policy as Read and Write access to the object’s value, with the additional restriction that read-only fields cannot be written to.

When the access policy denies a client’s request, the return value of the system call for the request is ERROR_PERMISSION_DENIED.

The root container of a SeaShell database is fixed when the database is created, but the mechanism by which the database is created is not specified in this document.

When an object is created, the value of its Owner metadata field is set to the same value as that of its parent container Owner metadata field. Ownership of an object can change. To change the ownership of an object, the value of the Owner metadata field can be set by calling the Sytem.Host.SetObject system call for the ‘@Owner’ virtual name of that object, provided that it is permitted under the access control rules.

Since it is impossible for a control program to access objects that are not owned by the same principal as the one whose identity it is running under, a control program needs to delegate access to ‘foreign’ objects to programs loaded from code modules that have the ability to run under the identity of the owner of the ‘foreign’ object. To do this, a control program MAY use the System.Host.SpawnVm, System.Host.CallVm and System.Host.ReleaseVm system calls. (See §4.7.2 in the Plankton specification for more details.)

8 References

[RFC2046]	*Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. N. Freed, Nathaniel Borenstein. November 1996. http://www.ietf.org/rfc/rfc2046.txt
[RFC2119]	S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, IETF RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt
[RFC3066]	H. Alvestrand. *Tags for the Identification of Languages*. http://www.ietf.org/rfc/rfc3066.txt.