The Kronos Grimoire

Greetings, young Wizard! A long and adventurous journey brought you to this source of secret knowledge and dark arts. With this grimoire as your faithful companion, you will uncover the deepest mysteries in the spiral.

What it is (not)

This cursed book is the primary source of documentation for the Kronos project, for regular users and contributors alike.

Topics that are within the scope of this book include, among others, documentation of the game's inner workings, descriptions of the game protocols, and file formats.

That being said, it is not a goal to discuss the algorithmic implementation details of both Kronos and official KingsIsle code.

Contributing

In the interest of access control and sensibility for what we share here, the sources of this book are hosted in a private repository with automated deployment to `kronos-project/grimoire`.

If you find any of the information confusing, misleading or wrong, or would like to suggest additional material and content to be covered here, don't hesitate to file an issue in the repository or reach out to `Vale#5252` on Discord.

While we do not accept direct contributions, exchange and suggestions are always welcome.

Licensing

Graphical Client Arguments

`WizardGraphicalClient` will happily spit out a list of CLI arguments if passed `-?` but this is outdated.

This page attempts to correctly document a list of available arguments.

Argument	Usage	Description
`-?`	`-?`	Print outdated arguments list
`-A`	`-A [STRING]`	Locale
`-CS`	`-CS`	Dumps the Client Signature
`-C`	`-C [STRING]`	Character name/ID
`-D`	`-D [PATH]`	Path to data root dir
`-EF_OVERFLOW`	`-EF_OVERFLOW`	Enable overflow detection
`-EF_UNDERFLOW`	`-EF_UNDERFLOW`	Enable underflow detection
`-G`	`-G [PATH]`	Path to log file
`-HD`	`-HD`	Enable heap debugging
`-HS`	`-HS`	Enable heap server
`-IgnoreMissingParams`	`-IgnoreMissingParams`	Ignore missing parameters
`-L`	`-L [HOST] [PORT]`	Login server to use
`-O`	`-O`	Whether to log all resource requests
`-PT`	`-PT [INT]`	"PatchClientPatchTime"
`-ST`	`-ST`	"-Steam Required"
`-T`	`-T [STRING]`	"Test Local Zone"
`-UN`	`-UN [0/1]`	Whether to force unique character names
`-U`	`-U ..[UID] [CK2] [USERNAME]`	User credentials passed in by patch client
`-V`	`-V`	"DoServerSelection"
`-X`	`-X`	"Dump Classes to Filename"
`-c`	`-c`	Checks compression for VolumeWAD files
`-u`	`-u [STRING],[STRING],...`	Wildcards for uncompressed VolumeWAD files

String ID

In many places throughout the game, strings are hashed into 32-bit integer values which are more compact to handle and convey similar intent.

Such places are *ObjectProperty* serialization, implementation-defined error codes in the network protocol, and enums in code.

Algorithm

The following presents our flavor of the algorithm which differs from KI's but produces matching results.

```python
def sign_extend(value: int) -> int:
    return (value & 0x7FFFFFFF) - (value & 0x80000000)


def make_string_id(string: str) -> int:
    result = 0

    for index, value in enumerate(string.encode()):
        value -= 32
        shift = 5 * index % 32

        result ^= sign_extend(value << shift)
        if shift > 24:
            result ^= sign_extend(value >> (32 - shift))

    return abs(result) & 0xFFFFFFFF
```

Data Management Layer

The "Data Management Layer", often referred to as "DML", is a proprietary data serialization system developed by KingsIsle for use in their games. At its core, it follows a remote procedure call model as it is primarily used for data exchange over the network which triggers the execution of designated handlers based on message types.

Protocols and data messages are described by a custom specification format in XML. Correspondingly, such specification files are often seen as regular `*Messages.xml` files. Their structure and composition will be explained on the fly when discussing protocols and messages.

Protocols

Protocols are the primary entity of the DML serialization system. Each protocol is uniquely identified by an ID and groups together several messages together.

One protocol may be described per XML file and the name of the protocol (which is equal to the specification file name in most cases) may be defined as the root XML tag:

```xml
<MyAwesomeProtocol>
  <!-- Protocol description here -->
</MyAwesomeProtocol>
```

Protocol information

Every protocol must have a child XML element named `_ProtocolInfo` assigned to it. Its record holds the following mandatory message fields:

Name	Type	Description
ServiceID	UBYT	A unique service ID for the protocol service provider
ProtocolType	STR	A unique name for the type of protocol being defined
ProtocolVersion	INT	The version of the DML system that is used by the protocol
ProtocolDescription	STR	A description of the purpose the protocol serves

When serializing protocol messages over the network, the ServiceID identifies the higher-order protocol of the message to read. As such, implementors will have to track all accessible `ServiceID`s accordingly for lookup.

For more information on types, records and fields, see the documentation for messages.

Example

```xml
<MyAwesomeProtocol>
  <_ProtocolInfo>
    <RECORD>
      <ServiceID TYPE="UBYT">123</ServiceID>
      <ProtocolType TYPE="STR">AWESOME</ProtocolType>
      <ProtocolVersion TYPE="INT">1</ProtocolVersion>
      <ProtocolDescription TYPE="STR">Super awesome protocol</ProtocolDescription>
    </RECORD>
  </_ProtocolInfo>
</MyAwesomeProtocol>
```

Messages

Messages are the central data structures exposed by a protocol. They define a data layout and are used for runtime serialization and deserialization of data.

A message consists of an XML element with its sole child element being a record holding the data fields along with their metadata.

A DML message may be defined like this:

```xml
<MSG_PING>
  <RECORD>
    <!-- Message metadata -->
    <_MsgName TYPE="STR" NOXFER="TRUE">MSG_PING</_MsgName>
    <_MsgType TYPE="UBYT" NOXFER="TRUE">1</_MsgType>
    <_MsgDescription TYPE="STR" NOXFER="TRUE">PING request.</_MsgDescription>
    <_MsgHandler TYPE="STR" NOXFER="TRUE">MSG_Ping</_MsgHandler>
    <_MsgAccessLvl TYPE="UBYT" NOXFER="TRUE">0</_MsgAccessLvl>

    <!-- Data fields -->
    <Count TYPE="UINT"></Count>
  </RECORD>
</MSG_PING>
```

Records

Records only ever occur as XML elements named `RECORD` inside protocol information elements and messages.

They group various fields together.

Fields

Fields are used to represent the data layout of a DML message structure or important metadata for both, protocols and messages alike. They only ever occur inside record elements.

When used to describe metadata, fields always have a fixed value assigned to them. In most cases however, their values are undefined and will be filled with runtime object values.

Field attributes

Fields may carry specific properties indicated by key-value pairs of XML attributes. The following presents known attributes, their supported values, and how they influence the behavior of a field.

Types

The most common attribute is `TYPE`. It specifies the data type of fields using a string acronym:

Name	Type	Description
BYT	int8	Signed 8-bit integer
UBYT	uint8	Unsigned 8-bit integer
USHRT	uint16	Unsigned 16-bit integer in little-endian byteorder
INT	int32	Signed 32-bit integer in little-endian byteorder
UINT	uint32	Unsigned 32-bit integer in little-endian byteorder
STR	uint8[]	A length-prefixed string of arbitrary bytes (may be UTF-8)
WSTR	uint16[]	A length-prefixed string of UTF-16 code points in little-endian order
FLT	float	IEEE-754 32-bit floating point number in little-endian byteorder
DBL	double	IEEE-754 64-bit floating point number in little-endian byteorder
GID	uint64	Unsigned 64-bit integer in little-endian byteorder

Visibility

Every field may individually control whether it is visible as a serializable data field or not. This is accomplished by setting the `NOXFER` attribute to either `"TRUE"` or `"FALSE"`.

NOXFER stands for "No Transfer" and is mostly used for metadata fields which carry fixed values. As per convention, such fields should always start with an underscore (`_`) in their name to highlight the hidden status.

Serialization

Serialization and deserialization of a message works by consecutively iterating through the fields in the order they were listed in the message specification, and encoding the value as defined by the type. Fields with the `NOXFER` attribute will be skipped.

Example

The message

```
<MSG_PERSON>
  <RECORD>
    <Name TYPE="STR">Edgar Allan Poe</Name>
    <Age TYPE="UBYT">40</Age>
  </RECORD>
</MSG_PERSON>
```

serializes to

```
[
    # String length prefix (15)
    0x0f, 0x00,
    # String bytes without null terminator (Edgar Allan Poe)
    0x45, 0x64, 0x67, 0x61, 0x72, 0x20, 0x41, 0x6c, 0x6c, 0x61, 0x6e, 0x20, 0x50, 0x6f,
0x65,
    # Age byte (40)
    0x28
]
```

Metadata

The records of messages may have a set of hidden fields assigned to them which solely describe metadata required for runtime handling and querying of individual messages.

Unlike the protocol information however, these fields must all be marked as `NOXFER`.

Below is a list of fields and their meaning. Some of them are discussed in greater detail in subsections of this paragraph.

Name	Type	Optional	Description
_MsgName	STR	true	The name of the message; Overrides the XML message tag
_MsgOrder	UBYT	true	The order value
_MsgDescription	STR	true	Short description of the message and its purpose
_MsgHandler	STR	false	The handler callback to process this message when received
_MsgAccessLvl	UBYT	true	The access level for the message

Order number

Every message has its own index assigned to it that is unique within a protocol - the order value. Order values usually start at `1` and are used in combination with the unique service ID of the protocol to address a specific message when several protocols are available at once.

As the value is represented as an unsigned 8-bit integer, the amount of messages per protocol is limited to 255 as a result of that.

Order values may either be explicitly specified using the `_MsgOrder` field or left away for every message within the protocol. In case of the latter, the implementation automatically assigns order values in ascending order by sorting message names (name from xml tag; NOT _MsgName) alphabetically. Mixing both approaches up may result in collisions!

Access level

The access level defines the minimum value that must be held by a session in order to be allowed to process the message.

For the scope of the system, this trait is fairly unimportant, but it plays a great role in handling network sessions and is further explained there.

DML Tables

This is a binary format which is used most notably for `LatestFileList.bin`.

All integers are in little-endian byte order unless otherwise specified.

Type Tag

When binary-serialized, the subsequent values of field types are identified by a tag value dynamically. The following maps tags to their respective DML types:

Tag	Type
0	GID
1	INT
2	UINT
3	FLT
4	BYT
5	UBYT
6	USHRT
7	DBL
8	STR
9	WSTR

Message Orders

DML tables at present only use message types defined in the `ExtendedBaseMessages` protocol.

Index	Type
1	MSG_CUSTOMDICT
2	MSG_CUSTOMRECORD

Structure

The following documents the outline of a serialized table blob.

Table

This structure and everything it encapsulates should be read as many times as needed until EOF is reached.

Name	Type	Description
ValuesLength	UINT	Length of the Values array
Values	Value[]	Array of Value structures

Value

Name	Type	Description
ProtocolID	UBYT	ID of `ExtendedBaseMessages` protocol (always 2)
-	UBYT	Type of structure that follows according to the Message Order
Size	USHRT	Size of this structure including everything it encapsulates

RecordTemplate

Name	Type	Description
RecordFields	RecordField[]	Array of RecordField structures

RecordField

Name	Type	Description
Length	USHRT	Length prefix of Name
Name	STR	Name of the field If the name is `_TargetTable` then the TargetTable structure should be read after this structure
Type	UBYT	Type of the field according to the Type Tags table
?	UBYT	DML flags; implementation-specific

TargetTable

This is assumed to *always* be part of a RecordTemplate as part of the specification.

Name	Type	Description
Length	USHRT	Length prefix of Name
Name	STR	The table that Records of this type should belong to

Record

Name	Type	Description
-	-	Values should be read according to the RecordFields in the preceding RecordTemplate

ObjectProperty

ObjectProperty is KingsIsle's runtime reflection and serialization system for C++ classes.

It was initially written by Richard Lyle as part of the Medusa project (see `Reflection` directory) and was later ported into KI's codebase with additional features and improvements.

The wizwalker project's type dumper utility may be used to obtain a full list of all reflected types from the game.

- Property Classes
- Serialization

Property Classes

Property Classes represent data structures with dynamic runtime reflection enabled on them, all of them inheriting from a common `PropertyClass` base.

On a high-level, they can be thought of as

```
class PropertyClass {
    /** Called before serialization. */
    virtual void OnPreSave() = 0;

    /** Called after serialization. */
    virtual void OnPostSave() = 0;

    /** Called before deserialization. */
    virtual void OnPreLoad() = 0;

    /** Called after deserialization. */
    virtual void OnPostLoad() = 0;
};
```

Properties

Properties are the reflected C++ members of a class. Note that these are not representative of the actual amount of members in total or their memory layout at all.

Notoriously, properties can either be enum types, bitflag types, or regular value types. They all have a name (which potentially differs from the actual C++ member name), a unique integer ID, and a set of configuration flags assigned.

Distinction between pointer types (raw and smart), references and value types is performed.

The subsections of discuss all the important attributes and their importance.

Names

It is valid to name properties after a nested path into the object it carries as a value.

Consider the following example:

```
union gid {
    unsigned __int64 m_full;
};
```

A property `gid m_id;` may in reality carry the name `m_id.m_full`, which indicates that instead of the `gid` serialization routine, the `unsigned __int64` routine should be used for the `m_full` member.

Such a `m_id.m_full` property would correspondingly also report `unsigned __int64` as its type instead of `union gid`.

Flags

Each property may carry a set of flag bits which influence its behavior. The following is a non-exhaustive list of known flag bits and their purpose:

Bit	Purpose
0	Property value can be saved
1	Property value may be copied/cloned
2	Property has public visibility
3	Property may be transmitted to players over the network
4	Property may be transmitted to privileged players over the network
5	Property may be persistently saved in database
6	The property is deprecated and must be skipped by the serializer
7	???
8	Property may only be re-serialized if modified
9	The `std::string` property stores a binary blob
16	Property must not be edited
17	`std::string`s holding file names
18	Set on properties of `class Color` type (some were forgotten)
20	C-style bit flag enum type
21	C++ enum class type where variants are scoped
22	`std::wstring`s holding i18n keys for GUI elements
23	`std::string`s holding string table key values
24	Property is a key for its containing class
25	Property is a key for a different class
27	Property that holds the name of its containing class
28	Set on properties that have the `__BASECLASS` option

Note that bits 16-28 are mostly editor hints without any semantic features to them.

Container

Every property further has a container type associated with it. Containers are dynamically-sized, homogenous collections of elements of the same type.

Containers are dynamically-sized, homogenous collections of elements of type `T`.

A property always reports the `T` as its type. Strictly speaking, a `List<std::string>` property actually has type `std::string` and the `"List"` container assigned to it.

The actual containers directly map onto C++ types:

Container	Property type	C++ type signature
`Static`	`T`	`T`
`Vector`	`T`	`std::vector<T>`
`List`	`T`	`std::list<T>`

Enum

Enums come in two variants:

- Plain, old C-style enums which are used as bit flag types.
- C++ enums with scoped variants

In both cases, the enum variants are runtime-accessible as options of variant name and value pairs.

Options

Options are pairs of names and values that can be assigned to a properties on an individual basis.

Most commonly, they occur with the aforementioned enum types, but they can also be used with other types to map commonly expected values to names.

Traditionally, they hold either integral or string values.

A handful of them have a special meanings associated with them. These are detailed in the following subsections.

Base classes

Some properties with `std::string` type have been observed to hold a special option named `__BASECLASS` which holds the name of a class type.

They are believed to be editor hints without any semantic impact.

Default values

Properties can be assigned default values using a special `__DEFAULT` option which holds a string representation of the default value a property should default to if no other value is given during initialization.

- To set an integer property to value 1 - `"__DEFAULT": "1"`
- To set a `Foo`-typed property to enum variant `Foo::kBar` - `"__DEFAULT": "kBar"`

Reflection

Reflection allows runtime interaction with property classes in one of the following ways:

- Introspection of values with unknown types
- Iterating over properties
- Querying properties by name/ID
- Accessing aforementioned property metadata
- Check if class is subclass of a certain other type
- Clone/copy property values into different types

Serialization

This aims to give insight into the serialization of Property Classes.

The following uses Python pseudocode for illustration which makes use of format strings defined by the struct module.

Binary

The binary serialization mode is commonly encountered for both local game files and also for `STR` types in many DML messages transferred over the network.

Buffering

For writing the binary data, a sink with bit-oriented instead of byte-oriented buffering is preferred due to some types being serialized in units of single bits only.

In such cases, the buffer should progress sequentially from the LSB of a byte to its MSB before advancing to the next one.

When the binary size of a type is in units of whole bytes, the buffer will be aligned to the start of a full byte with bit position 0, if not already there, before writing said type.

Flags

Binary serializers and deserializers may have a set of flags attached to them to customize their behavior:

Bit	Purpose
0	Indicates that these flags should be serialized and re-used by the deserializer
1	Tries to pack length prefixes into smaller quantities for compact serialization
2	Causes enum variants to be serialized as human-readable strings instead of values
3	Enables zlib compression of serialized object state
4	Properties with flag bit `8` set must always be dirty when serialized

Header

A serialized stream starts with the necessary header data followed by the compressed or uncompressed object bytes:

```
output = bytearray()

# Serialize our flags value if `STATEFUL_FLAGS` (bit 0) is set.
if serializer_flags & STATEFUL_FLAGS != 0:
    output.extend(serializer_flags.to_bytes(4, "little"))

# Handle compression if `WITH_COMPRESSION` (bit 3) is set.
if serializer_flags & WITH_COMPRESSION != 0:
    compressed_object_data = zlib.compress(object_data)

    if len(compressed_object_data) < len(object_data):
        object_data = compressed_object_data

        # Indicate that the data is compressed.
        output.append(1)
        # Write the size of the uncompressed object for the deserializer to validate.
        output.extend(len(object_data).to_bytes(4, "little"))
    else:
        # Indicate that the data is uncompressed.
        output.append(0)

# Write either the compressed or uncompressed data.
output.extend(object_data)
```

Objects and Properties

The serialization system deals with whole `PropertyClass` es at any time. No loose values anywhere.

A serializer accepts the following inputs to customize its behavior:

- a mask of serializer flags for configuration
- a boolean denoting whether the output will be *shallow* or *deep*
- a wildcard of property flag bits to only serialize those properties where that mask is an intersection of the actual flags

Data model

The *ObjectProperty* data model defines what types are supported and how they are serialized. This may be freely extended with custom types that are implementation-defined and not `PropertyClass` es themselves.

The following examples of serialization modes will use an imaginary `serialize_value` function that should be thought of as a mapping arbitrary values into this data model and serializing them to the `buffer` argument.

Be sure to consider the buffering remarks at the start when implementing this.

- booleans will be written as a single bit; `1` for `true` and `0` for `false`
- primitive integer types (signed and unsigned) will be written as bytes in little-endian order
- floating-point numbers according to IEEE-754 are bit-copied into `uint32_t` / `uint64_t` and serialized as such
- strings are serialized as UTF-8 bytes with their **length prefixed**
- wide strings are serialized as UTF-16 code points in little-endian order without BOM and with their **length prefixed**
- collections, such as lists or vectors, are serialized as a sequence of element values with their **length prefixed**
- tuples or arrays with a known length are serialized as just the sequence of elements
- when a property is **opional** (i.e. has bit 8 set in its flags set), its value may be skipped (unless the serializer has bit 4 set in its flags); a single bit of `0` denotes that no value is given, otherwise a value of `1` followed by the property's value is written
- enum variants are either serialized as their integral value or, when serializer flag bit 2 is set, as a string representation of the variant name
  - an empty string for bit enums is equivalent to a value of `0`
  - the bit enum variant string is a list of flag names: `A|B|C`
- **length prefixes** are `uint16_t` for (w)strings and `uint32_t` for collections unless serializer bit 1 is set, which enables a common compression algorithm applied to both types - when the length is smaller than `0x80`, write it as `uint8_t` with the LSB set to `0`, otherwise write it as `uint32_t` with LSB set to `1`

Type Tag

Every serialized `PropertyClass` state has a type tag associated with it to uniquely identify it during deserialization.

The type tag is a **string ID** of the type's name.

Property Tag

Property tags uniquely identify a property within an object in *deep* serialization mode.

The tag is a sum of the property type's **string ID** and a slightly modified djb2 hash of the property's name with the MSB value discarded.

Practically speaking:

```
type_tag = string_id(property.type_name)  # NOT object.type_name
name_hash = djb2(property.name) & 0x7FFF_FFFF

property_tag = (type_tag + name_hash) & 0xFFFF_FFFF
```

Shallow mode

In shallow mode, the 32-bit object type tag is written followed by a sequence of masked property values in their correct order:

```
buffer = BinaryBuffer()

buffer.write("<I", object.type_hash)
for property in filter(lambda p: p.flags & mask == mask, object.properties):
    serialize_value(buffer, property.value)
```

This mode is **not** allowed to skip properties with the `DEPRECATED` (bit 6) flag set, as a correct order of values is the only indicator that exists to correctly reconstruct the object during deserialization.

Deep mode

In deep mode, the concept is a bit different. Here, the 32-bit object type tag is serialized, followed by a mapping of property tags to their values. Additionally, size information in bits is written for integrity validation.

In practice, this looks like this:

```python
buffer = BitBuffer()

buffer.write("<I", object.type_hash)

# Reserve a placeholder for the object size.
object_size_position = len(buffer)
buffer.write("<I", 0)

# Here we don't only skip unmasked properties, but also deprecated ones.
for property in filter(
    lambda p: p.flags & mask == mask and p.flags & FLAG_DEPRECATED == 0,
object.properties
):
    # Reserve a placeholder for the property size.
    property_size_position = len(buffer)
    buffer.write("<I", 0)  # Will be replaced by a real size later.

    # Write the mapping of property hash to value.
    buffer.write("<I", property.hash)
    serialize_value(buffer, property.value)

    # Patch back the real property size.
    buffer.seek_bit(property_size_position)
    buffer.write("<I", len(buffer) - property_size_position)

# Patch back the real object size.
buffer.seek_bit(object_size_position)
buffer.write("<I", len(buffer) - object_size_position)
```

The order of property entries, while usually maintained, is not as important as it is for **shallow** serialization.

Files

When serializing to files, a common convention is to use an `.xml` suffix. This orignates from different ways of representing the serialized data inside them.

For debugging purposes, a human-readable format is often desired. It is very straightforward and can be fully explained in a short example:

```xml
<Objects>
  <Class Name="class Example">
    <!-- We place a tag for every property and its value as the tag's content. -->
    <m_someString>Test</m_someString>
    <m_someInt>1337</m_someInt>
    <m_someObject>
      <Class Name="class SomeObject">
        <m_test>Properties holding objects will hold a nested Class element</m_test>
      </Class>
    </m_someObject>
    <m_someTuple>1,0,0,1</m_someTuple>

    <!-- This is how we serialize properties holding container values. -->
    <m_listOfStrings>A</m_listOfStrings> <!-- Index 0 -->
    <m_listOfStrings>B</m_listOfStrings> <!-- Index 1 -->
    <m_listOfStrings>C</m_listOfStrings> <!-- Index 2 -->
  </Class>
</Objects>
```

When distributing game data, specifically data that is not meant to be edited afterwards, a more compact format is often preferred. This is exhaustive binary serialization with a special file magic:

```python
FILE_MAGIC = 0x644E4942  # b"BINd" in little-endian byteorder

buffer.write("<I", FILE_MAGIC)
buffer.extend(serialized_object_state)
```

Network Protocol

The game makes use of a custom application-level protocol on top of TCP/UDP.

This section goes into great detail of said protocol regarding its creation and management of user sessions and the custom framing protocol it uses to exchange data.

Note: Readers should be familiar with the Data Management Layer system prior to working through this.

Framing

At its core, a custom data framing format is used to exchange messages between two peers maintaining a connection. No difference is made between TCP-based and UDP-based connections.

Frames generally start with a header, followed by the body holding the frame opcode and the encapsulated message payload. Assume all data to be encoded in little-endian byteorder unless stated otherwise.

Header

Depending on the header type being used, the offsets of data in the following body may be shifted.

Small

For data messages where the DML message payload is below `0x8000` bytes in size, the following header is used:

Offset	Type	Description
0x0	uint16	The constant magic `0xF00D`
0x2	uint16	The length of the following body

Large

For data messages above `0x7FFF` bytes in length, a different header encoding strategy is employed:

Offset	Type	Description
0x0	uint16	The constant magic `0xF00D`
0x2	uint16	The constant `0x8000`
0x4	uint32	The real size of this "large" frame

A deserializer should thus determine the size of the frame based on whether the second header field value is `>= 0x8000`.

Body

The body provides metadata of the contained message payload that is needed to decode it.

Messages are either control messages or data messages which are further detailed in their respective sections.

Offset	Type	Description
0x0	bool	Whether the frame is a control message
0x1	uint8	The message opcode; only for control messages
0x2	uint8	Reserved; always zero
0x3	uint8	Reserved; always zero
0x4	uint8[]	The contained message data

Control Messages

Control messages are transmitted to signal events related to the creation and maintenance of sessions and connections in general.

The following explains the purpose and structure of various types of control messages.

TODO: Figure out opcode 0x1 and 0x2.

Opcode

To identify a control message in the first place, the frame body encodes the message opcode. Opcodes are unique values that map to exactly one type of frame and tell the parser how the message data should be interpreted.

For concrete values, see the following message types.

Session Offer

> Opcode: `0x0`

When a new client connects to the server, it is greeted with this type of frame. Since most of the game data exchange requires an active session, it must first be established through the client completing the handshake by responding with a Session Accept message.

Offset	Type	Description
0x0	uint16	The proposed session ID to agree on
0x2	int32	The high 4 bytes of the UNIX timestamp the message was sent at
0x6	int32	The low 4 bytes of the UNIX timestamp the message was sent at
0xA	uint32	The milliseconds into the second this message was sent at
0xE	uint32	Length prefix for unknown field
0x12	uint8[]	Unknown
-	uint8	Reserved; always zero

Keep Alive

Opcode: `0x3`

A bi-directional opcode that may be initiated independently by both, client and server. These messages are exchanged at fixed intervals and a Keep Alive Rsp from the other peer ensures the session is still alive.

The structure of this message (and the response to it) varies depending on the party sending it:

Client-initiated Keep Alive:

Offset	Type	Description
0x0	uint16	The corresponding session ID
0x2	uint16	The milliseconds into the second the message was sent at
0x4	uint16	How many minutes have elapsed since the session was started

Server-initiated Keep Alive:

Offset	Type	Description
0x0	uint16	Invalid session ID value
0x2	uint32	The number of milliseconds since the server was started

Keep Alive Rsp

Opcode: `0x4`

This message is used to acknowledge a received Keep Alive message and confirm that the other end of the connection is still listening. Either peer should respond to this using the structure of the preceding Keep Alive message.

Session Accept

Opcode: `0x5`

Sent from the client to the server as a response to a Session Offer message. This completes the handshake and confirms the creation of a new session bound to the proposed ID.

From that point onwards, the session must be kept alive by heartbeating.

Offset	Type	Description
0x0	uint16	Reserved; always zero
0x2	int32	The high 4 bytes of the UNIX timestamp the message was sent at
0x6	int32	The low 4 bytes of the UNIX timestamp the message was sent at
0xA	uint32	The milliseconds into the second the message was sent at
0xE	uint16	The proposed session ID from Session Offer
0x10	uint32	Length prefix for unknown field
0x14	uint8[]	Unknown
-	uint8	Reserved; always zero

Data Messages

Data messages carry application-specific data defined by a DML Protocol. However, DML as a serialization system is not inherently related to the framing protocol itself, and should be considered a separate entity which is chosen due to its flexibility.

Along with the encoded DML payload, a small header provides the metadata needed for the peer to understand the message.

The "message" terminology can either stand for a data message as part of the framing protocol, which is documented here, or a DML message as a data structure encoded inside a data message. These are not to be confused with each other, see Data Management Layer for more details on the serialization system.

Structure

An encoded message always follows a fixed structure. The opcode in the frame body is always set to `0` for data messages.

Offset	Type	Description
0x0	uint8	The service ID the message belongs to
0x1	uint8	The order number within the protocol
0x2	uint16	The length of the entire DML message data, including this header
0x4	uint8[len]	The serialized DML message data
`len`	uint8	Trailing null byte; not considered by the length field

Sessions

Sessions between a client and a server are an important aspect of the network communication as they grant various privileges and confirm the authenticity of a connecting clients.

Handshake

To establish a new sessions between two peers, either directly after a client opened a connection or after the invalidation of a session, a handshake must be performed.

Clients should listen for Session Offer messages at any time, and respond with the appropriate Session Accept after receiving it.

When the session IDs match and the server does not respond with yet another Session Offer message, the session should be considered established.

Both parties are expected to cache the relevant information from the handshake, such as server uptime values, the timestamp the messages were sent, and obiously also the ID. Various algorithms may rely on these data as unique variables individually assigned to every client.

Refreshing a session

During the lifetime of a TCP or UDP connection with the server, a very common event to expect is such sessions actually being dropped in the middle of operations.

This act of dropping sessions is generally taking place when a client connects to several servers at the same time and establishes sessions with them while already maintaining a session with one of the servers. The server in question then tells all other servers (including the ones the client is still connected to) to drop any sessions with that specific client.

But instead of closing the underlying socket, the session will be refreshed. Since servers are assuming that active exchange of messages only happens with one of them at the same time, regardless of how many a client is actually connected to, a server will wait for the first message from a client after the session invalidation, ignore it, and instead initiate the handshaking process with a different ID.

When the process is completed, the client is expected to re-send its last message prior to receiving a new Session Offer message as the server ignored it up to that point.

Session ID

The session ID is a unique identifier assigned to every active session individually. It is represented as an unsigned 16-bit integer and only one connection may occupy any valid session ID at the same time.

A value of `0` is reserved to the server to indicate that a Keep Alive message is server-initiated. It should never be assigned to clients in order to avoid confusion of frame types.

Heartbeat

When the handshake was successfully completed and the session is up and running, the heartbeating process must begin in order to keep the session alive.

This step involves exchanging Keep Alive and corresponding Keep Alive Rsp messages at fixed intervals and waiting for the response of the peer.

Clients send such a request every **10 seconds**, whereas servers independently do so every **60 seconds**.

When no response had been received directly before sending another request, the connection is considered dead and the underlying socket should be closed.

Access Level

Access Levels are numeric values denoting a specific set of permissions associated with the session between an individual client and the server. Generally speaking, the higher the value, the higher the permissions it grants.

Every DML message may have an access level value specified as a minimum threshold that must be met in order for a peer to be allowed to process the data.

A non-exhaustive list of known values and their meanings is:

Value	Description
`0`	No special requirements; can be used anytime
`1`	A valid session must be established in order to process the message

KIWAD Archives

The "KingsIsle Where's All the Data?" format (name guessed) is a custom file format used by the game to bundle all kinds of game assets in archives. The format presumably draws inspiration from the original Doom WAD format and similarly uses `.wad` as file extension.

Archives consist of a header, followed by the file table, and lastly the encoded file data as described in the table. All data is assumed to be encoded in little-endian byteorder unless stated otherwise.

Header

The header may consume 13 or 14 bytes in size and outlines the amount of files to expect in the archive:

Offset	Type	Description
0x0	string	`"KIWAD"` as an ASCII string
0x5	uint32	The KIWAD version that is used
0x9	uint32	An integer holding the amount of files in the archive
0xD	uint8	optional; WAD Flags if version >= `2`

Flags

On file format versions >= 2, the header encodes a mask of configuration bits which should be treated as implementation-defined for the most parts; KIWAD implementations can safely ignore these bit flags and always write `0`.

This is how the official client uses them:

Bit	Name
0	Memory-maps the file view
1	Prefetches the file handle

File table

This table consists of `n` repetitions of the file structure, where `n` is the number of archived files as encoded in the header.

File

This structure denotes the metadata of an encoded file, needed for extraction and validation:

Offset	Type	Description
0x0	uint32	The start offset of the file data in the archive
0x4	uint32	The size of the uncompressed file in bytes
0x8	int32	The size of the compressed file in bytes
0xC	bool	Whether the encoded file data is compressed
0xD	uint32	The checksum of the file data
0x11	uint32	The length of the file's path in the archive in bytes
0x15	char[]	The archive file name as a null-terminated string

Compression

If a file is compressed, the corresponding file table entry encodes its uncompressed size and the compressed size and the data will run through zlib.

If a file is stored uncompressed, `-1` (or `0xFFFF_FFFF`) will be written for its compressed size.

Notably, only `.mp3` and `.ogg` files are shipped uncompressed in official archives by KingsIsle.

Checksums

The checksum of a file is calculated by CRC32 using the polynomial `0x4C11DB7` and reversed input and output values:

```python
import crcengine

crc32 = crcengine.create(0x04C11DB7, 32, 0, xor_out=0)

assert crc32(b"KIWAD") == 4265429514
```

Login Server

This chapter describes the architecture and functionality of the *Login Server*.

Its purpose is to authenticate players, perform *Game Server* load balancing, and manage in-game characters for the account.

The primary data exchange protocol is described in DML as `LoginMessages.xml`, Service ID `7`. It is recommended to keep this open for reference while reading through the following pages.

Groundwork

This discusses the foundations and prerequirements for a Login Server implementation without going into detail on concrete functionality yet.

Client Connectivity

In practical terms, the timeframe of communication between game client and *Login Server* starts with opening the Patch Client/WizardGraphicalClient and ends when hitting the **Play** button in character selection.

The Patch Server will source the server IP and port from `PatchConfig.xml` and forward the information to the WizardGraphicalClient via the `-L` CLI argument.

> When trying to make the Patch Client connect to a custom Login Server, it is mandatory to bypass KingsIsle's *Patch Server* or the `PatchConfig.xml` change will be reverted before being able to make it into the game.

Secondary Message Protocols

The Login Server fundamentally needs to support some basic functionality aside from its primary `LoginMessages` protocol.

Pings

Every so often, the client may send `MSG_PING` from `BaseMessages.xml` (Service ID `1`) to the server to check its responsiveness. The server should respond with `MSG_PING_RSP` from the same protocol upon receiving. Both these messages don't carry any data payloads.

Client Disconnection

When clients shut down gracefully, they send `MSG_CLIENT_DISCONNECT` from `GameMessages.xml` (Service ID `5`). In such an event, the server should close the open client connection and free up occupied resources.

Note that a variety of problems on the user end can summon unexpected connection aborts. The implementation should still detect these events and free up its resources accordingly.

Client Sessions

When a client initially connects to the *Login Server*, a new session must be established before being able to process any `LoginMessages`.

A server implementation should cache **Session ID** and **Session Offer timestamp** per client as they will be needed later.

Authentication

With a valid session established, users will enter their usernames and passwords and hit the **Login** button in order to have their identity verified.

This is done through the `MSG_USER_AUTHEN_V3` request that will be crafted and sent by clients.

ClientKey1

ClientKey1, commonly abbreviated as CK1, is a string generated by the client as a hash of the user's password tied to the current session to avoid data replay from captured authentication requests.

The algorithm goes as follows, with **password** being the user's plaintext password input, **sid** being the cached Session ID for the connection, **time_secs** and **time_millis** being seconds since epoch and subsecond millis extracted from the cached Session Offer timestamp, respectively:

```python
from base64 import b64encode as base64
from hashlib import sha512

# Produce a base64-encoded SHA512 hash of the password and hash that again.
state = sha512(base64(sha512(password).digest()))
# Mix in salt built from previously cached session information.
state.update(f"{sid}{time_secs}{time_millis}")

# Receive CK1 as the base64-encoded hash of *password hash* and *salt*.
clientkey1 = base64(state.digest())
```

REC1 (serverbound)

`REC1` is a field in the aforementioned `MSG_USER_AUTHEN_V3` data message that holds a record of the user's credentials.

The actual data there is encrypted with Twofish in OFB mode.

The logic used to derive Twofish keys and IVs for this operation is:

```go
func generateIV() []byte {
    const ivConstant = 0xB6

    iv := make([]byte, 16)
    for i := 0; i < len(iv); i++ {
        iv[i] = ivConstant - byte(i)
    }

    return iv
}

func generateKey(sid uint16, timeSecs uint32, timeMillis uint32) []byte {
    const keyConstant = 0x17

    key := make([]byte, 32)
    for i := 0; i < len(key); i++ {
        key[i] = keyConstant + byte(i)
    }

    le := make([]byte, 4)

    binary.LittleEndian.PutUint16(le, sid)
    key[4] = le[0]
    key[5] = le[2] // This is always zero
    key[6] = le[1]

    binary.LittleEndian.PutUint32(le, timeSecs)
    key[8] = le[0]
    key[9] = le[2]
    key[12] = le[1]
    key[13] = le[3]

    binary.LittleEndian.PutUint32(le, timeMillis)
    key[14] = le[0]
    key[15] = le[1]

    return key
}
```

With this at hand, the actual Record can be built and encrypted:

```go
func buildRecord1(username string, ck1 string, sid uint16, timeSecs uint32, timeMillis
uint32) []byte {
    key := generateKey(sid, timeSecs, timeMillis)
    iv := generateIV()

    // Prepare the Twofish OFB context for later encryption
    block, _ := twofish.NewCipher(key)
    stream := cipher.NewOFB(block, iv)

    // Build the plaintext Record we would like to send
    record := fmt.Sprintf("%v %v %v", sid, username, ck1)

    // Encrypt and return the record
    stream.XORKeyStream(record, record)
    return record
}
```

Servers will need to decrypt the received record, split the plaintext at whitespace and parse it as `Session ID, Username, ClientKey1` which can be accordingly validated by deriving the CK1 for the password hash stored along with the username.

REC1 (clientbound)

After successful validation of an authentication request, the server is expected to respond with a `MSG_USER_AUTHEN_RSP` message.

This contains the unique `UserID` for the account, a `PayingUser` boolean denoting whether the account has active membership, and yet another `Rec1` string.

It is Twofish OFB encrypted as well and uses the same keys and IVs as showcased above. Clients should use the same logic for decrypting that string.

The Record is yet another base64-encoded string of 64 bytes, referred to as ClientKey2. The exact algorithm in use on KI's serverside remains unknown.

Note that ClientKey2 is, in itself, a secret value. It is session-agnostic and the mere knowledge of it is proof of identity to the server that will be validated when entering the game. Implementors are encouraged to use a suitable CSPRNG to generate random data and invalidate actively cached CK2s in fixed intervals, forcing users to re-authenticate.

When everything went smoothly, WizardGraphicalClient will be launced with the `-U` flag.

Errors

Many different errors may occur during authentication. For that purpose, `MSG_USER_AUTHEN_RSP` features an `Error` field which holds a String ID.

Below is a non-exhaustive list of known codes and when they should be sent:

String	When to send
`""`	No error
`"AccountBanned"`	User's account is banned
`"MachineBanned"`	User's machine is banned
`"AuthenFailed"`	Invalid credentials or internal server error
`"AISNoLogin"`	Chinese Anti-Indulgence System (legacy)
`"Timeout"`	Timeout while trying to process the request
`"FtpCapped"`	???
`"ErrorNoLock"`	???
`"FailedUpload"`	???

Legacy Authen

The login protocol also features `MSG_USER_AUTHEN` and `MSG_USER_AUTHEN_V2` . They are no longer supported and server implementations must ignore these requests.

Game Transition

This is a continuation of the Authentication Chapter.

After successful exchange of `MSG_USER_AUTHEN_V3` and `MSG_USER_AUTHEN_RSP` messages, the graphical client is now up and running in posession of the player's `UserID`, username and `ClientKey2`.

The next step is the transition from Patch Client to WizardGraphicalClient.

PassKey3

PassKey3, or PK3, is derived from ClientKey2 by the client and will be sent to the server. It uses the same algorithm as ClientKey1:

```python
from base64 import b64encode as base64
from hashlib import sha512

# Feed ClientKey2 into the hasher.
state = sha512(clientkey2)
# Mix in salt built from currently cached session information.
# NOTE: These may NOT be the same values which were used for authentication.
state.update(f"{sid}{time_secs}{time_millis}")

# Receive PK3 as the base64-encoded hash of *ClientKey2* and *salt*.
passkey3 = base64(state.digest())
```

Identity Validation

Request

The game client first sends `MSG_USER_VALIDATE` with the previously obtained `UserID` and `PassKey3`.

The same `MachineID` and `PatchClientID` that were already used in `MSG_USER_AUTHEN_V3` should be sent to reduce the possibility of account theft if machines were switched in-between authentication and validation. Servers should do the mandatory checks.

Response

The server is then supposed to respond with `MSG_USER_VALIDATE_RSP`. This once again contains an echo of the `UserID`, `PayingUser` and a potential error code.

Errors

The error codes are once again string IDs with known values listed below:

String	When to send
`""`	No error
`"AccountBanned"`	User's account is banned
`"MachineBanned"`	User's machine is banned
`"ValidateFailed"`	Invalid PK3 for user or internal server error
`"Timeout"`	Timeout while trying to process the request
`"FtpCapped"`	???
`"ErrorNoLock"`	???
`"FailedUpload"`	???

Load Balancing

Busy Queue

Immediately after `MSG_USER_VALIDATE_RSP`, the server is expected to follow up with `MSG_USER_ADMIT_IND`.

This is a load balancing mechanism with the *Game Server* to ensure not too many users are trying to enter the game at the same time.

The server is expected to maintain a FIFO queue of users trying to enter the game and admit them one at a time when the server can currently handle the load.

When the queue positions of users change, they are supposed to be informed with `MSG_USER_ADMIT_IND` every time. The user that was removed from the queue is position `0`.

Status

Value	Meaning
`1`	Success

Afk Handling

The server is expected to kick AFK players at fixed intervals to prevent them from taking up server resources unnecessarily.

Each time the player clicks a button in the WizardGraphicalClient, a `MSG_LOGIN_NOT_AFK` message should be sent to reset the kick interval for the current connection.

When the interval expires, the server should free all resources, send a user-facing notification, and close the connection.

Character Management

After successful authentication, validation and admission to the game, players will gain access to the character management scene.

WizardCharacterCreationInfo

`WizardCharacterCreationInfo` is a `PropertyClass` used to represent character data throughout early character selection.

As explained in the ObjectProperty chapter, readers should use wizwalker to obtain a dump of all types (including this one).

`m_schoolOfFocus` is a string ID of the school's literal name and `m_nameIndices` is a concatenations of three bytes for each index with the MSB always being `0`. Everything else is self-explanatory.

It is serialized with no serializer configuration flags (value `0`) and a property mask of `0x18`.

Requesting the List

The Character List must be directly requested from the server when initially entering the game but also when altering it, e.g. creating new characters.

The client sends the empty `MSG_REQUESTCHARACTERLIST` for that.

The server then follows up with `MSG_STARTCHARACTERLIST` holding a human-readable name of the *Login Server* instance and the number of additionally purchased character slots (to the default 6).

Next, `MSG_CHARACTERINFO` will be sent a finite number of times for each character there is to this account. It carries a serialized `WizardCharacterCreationInfo` object.

Finally, the sequence will be terminated with `MSG_CHARACTERLIST` holding an error code of `0` (empty string id). This message may be sent prematurely if an error occurs while still trying to stream the Character List.

Creating new Characters

The client sends `MSG_CREATE_CHARACTER` with the serialized `WizardCharacterCreationInfo` object.

The server then responds with `MSG_CREATE_CHARACTER_RESPONSE`, potentially holding an error or empty string ID on success.

Errors may occur when the creation info object is malformed or the character limit for the account is exhausted.

> TODO: Brute force string for `1745079928`.

Creation Logs

The client logs various stages of the character creation in `MSG_LOGIN_LOG_CHARACTER_CREATION`. The following explains the `Stage`s and `Parameter`s in use there.

Stage	Description	Parameter
`0`	Prologue	`0` (nothing)
`1`	Consulting Book of Secrets	`0` (nothing)
`2`	School chosen arbitrarily	School ID
`3`	School assigned by Book	School ID
`4`	Default appearance taken	Template ID
`5`	Custom appearance chosen	Template ID
`6`	Choosing a name	`0` (nothing)

School ID

Value	School
`0`	Fire
`1`	Ice
`2`	Storm
`3`	Myth
`4`	Life
`5`	Death
`6`	Balance

Deleting a Character

The client sends `MSG_DELETE_CHARACTER` with the unique `CharID` of the character.

The server is expected to respond with `MSG_DELETE_CHARACTER_RESPONSE`, potentially holding an error or empty string ID on success.

Errors may occur when `CharID` does not exist or references a character that does not belong to the currently authenticated account.

Entering the Game

When the player selected its character of choice and hits **Play**, the client will send `MSG_SELECTCHARACTER` with the selected character ID.

The server will then confirm the selection with `MSG_CHARACTERSELECTED` with IP and port of the *Game Server*, a random Key that will be sent back for validation later, player and char IDs, the current zone and position, and a potential error.

New Wizards start at `WizardCity/Tutorial_Exterior`, with zone ID `191965934121493239` and location `"START"`.

From this point onwards, the *Login Server* connection is closed and all client communication will happen through the *Game Server*.