The Move Book

Binary Canonical Serialization (BCS) is a binary encoding format for structured data. It was originally designed in Diem, and became the standard serialization format for Move. BCS is simple, efficient, deterministic, and easy to implement in any programming language.

The full format specification is available in the BCS repository .

BCS is a binary format that supports unsigned integers up to 256 bits, options, booleans, unit (empty value), fixed and variable-length sequences, and maps. The format is designed to be deterministic, meaning that the same data will always be serialized to the same bytes.

"BCS is not a self-describing format. As such, in order to deserialize a message, one must know the message type and layout ahead of time" from the README

Integers are stored in little-endian format, and variable-length integers are encoded using a variable-length encoding scheme. Sequences are prefixed with their length as ULEB128, enumerations are stored as the index of the variant followed by the data, and maps are stored as an ordered sequence of key-value pairs.

Structs are treated as a sequence of fields, and the fields are serialized in the order they are defined in the struct. The fields are serialized using the same rules as the top-level data.

The Sui Framework includes the sui::bcs module for encoding and decoding data. Encoding functions are native to the VM, and decoding functions are implemented in Move.

To encode data, use the bcs::to_bytes function, which converts data references into byte vectors. This function supports encoding any types, including structs.

The following example shows how to encode a struct using BCS. The to_bytes function can take any value and encode it as a vector of bytes.

Structs encode similarly to simple types. Here is how to encode a struct using BCS:

Because BCS does not self-describe and Move is statically typed, decoding requires prior knowledge of the data type. The sui::bcs module provides various functions to assist with this process.

BCS is implemented as a wrapper in Move. The decoder takes the bytes by value, and then allows the caller to peel off the data by calling different decoding functions, prefixed with peel_* . The data is split off the bytes, and the remainder bytes are kept in the wrapper until the into_remainder_bytes function is called.

There is a common practice to use multiple variables in a single let statement during decoding. It makes code a little bit more readable and helps to avoid unnecessary copying of the data.

While most of the primitive types have a dedicated decoding function, vectors need special handling, which depends on the type of the elements. For vectors, first you need to decode the length of the vector, and then decode each element in a loop.

For most common scenarios, bcs module provides a basic set of functions for decoding vectors:

Option is represented as a vector of either 0 or 1 element. To read an option, you would treat it like a vector and check its length (first byte - either 1 or 0).

If you need to decode an option of a custom type, use the method in the code snippet above.

The most common scenarios, bcs module provides a basic set of functions for decoding Option's:

Structs are decoded field by field, and there is no standard function to automatically decode bytes into a Move struct, and it would have been a violation of the Move's type system. Instead, you need to decode each field manually.

Binary Canonical Serialization is an efficient binary format for structured data, ensuring consistent serialization across platforms. The Sui Framework provides comprehensive tools for working with BCS, allowing extensive functionality through built-in functions.

Format

BCS is a binary format that supports unsigned integers up to 256 bits, options, booleans, unit (empty value), fixed and variable-length sequences, and maps. The format is designed to be deterministic, meaning that the same data will always be serialized to the same bytes.

"BCS is not a self-describing format. As such, in order to deserialize a message, one must know the message type and layout ahead of time" from the README

Integers are stored in little-endian format, and variable-length integers are encoded using a variable-length encoding scheme. Sequences are prefixed with their length as ULEB128, enumerations are stored as the index of the variant followed by the data, and maps are stored as an ordered sequence of key-value pairs.

Structs are treated as a sequence of fields, and the fields are serialized in the order they are defined in the struct. The fields are serialized using the same rules as the top-level data.

The Sui Framework includes the sui::bcs module for encoding and decoding data. Encoding functions are native to the VM, and decoding functions are implemented in Move.

To encode data, use the bcs::to_bytes function, which converts data references into byte vectors. This function supports encoding any types, including structs.

```
bash // File: move-stdlib/sources/bcs.move public native fun to_bytes<T>(t: &T): vector<u8>;
```

The following example shows how to encode a struct using BCS. The to_bytes function can take any value and encode it as a vector of bytes.

```bash use sui::bcs;

// 0x01 - a single byte with value 1 (or 0 for false) let bool_bytes = bcs::to_bytes(&true); // 0x2a - just a single byte let u8_bytes = bcs::to_bytes(&42u8); // 0x2a00000000000000 - 8 bytes let u64_bytes = bcs::to_bytes(&42u64); // address is a fixed sequence of 32 bytes // 0x0000000000000000000000000000000000000000000000000000000000000002 let addr = bcs::to_bytes(&@sui); ```

Structs encode similarly to simple types. Here is how to encode a struct using BCS:

```bash let data = CustomData { num: 42, string: b"hello, world!".to_string(), value: true };

let struct_bytes = bcs::to_bytes(&data);

let mut custom_bytes = vector[]; custom_bytes.append(bcs::to_bytes(&42u8)); custom_bytes.append(bcs::to_bytes(&b"hello, world!".to_string())); custom_bytes.append(bcs::to_bytes(&true));

// struct is just a sequence of fields, so the bytes should be the same! assert!(&struct_bytes == &custom_bytes); ```

Because BCS does not self-describe and Move is statically typed, decoding requires prior knowledge of the data type. The sui::bcs module provides various functions to assist with this process.

BCS is implemented as a wrapper in Move. The decoder takes the bytes by value, and then allows the caller to peel off the data by calling different decoding functions, prefixed with peel_* . The data is split off the bytes, and the remainder bytes are kept in the wrapper until the into_remainder_bytes function is called.

```bash use sui::bcs;

// BCS instance should always be declared as mutable let mut bcs = bcs::new(x"0100000000000000");

// Same bytes can be read differently, for example: Option let value: Option = bcs.peel_option_u64();

assert!(value.is_some()); assert!(value.borrow() == &0);

let remainder = bcs.into_remainder_bytes();

assert!(remainder.length() == 0); ```

There is a common practice to use multiple variables in a single let statement during decoding. It makes code a little bit more readable and helps to avoid unnecessary copying of the data.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// mind the order!!! // handy way to peel multiple values let (bool_value, u8_value, u64_value) = ( bcs.peel_bool(), bcs.peel_u8(), bcs.peel_u64() ); ```

While most of the primitive types have a dedicated decoding function, vectors need special handling, which depends on the type of the elements. For vectors, first you need to decode the length of the vector, and then decode each element in a loop.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// bcs.peel_vec_length() peels the length of the vector :) let mut len = bcs.peel_vec_length(); let mut vec = vector[];

// then iterate depending on the data type while (len > 0) { vec.push_back(bcs.peel_u64()); // or any other type len = len - 1; };

assert!(vec.length() == 1); ```

For most common scenarios, bcs module provides a basic set of functions for decoding vectors:

Option is represented as a vector of either 0 or 1 element. To read an option, you would treat it like a vector and check its length (first byte - either 1 or 0).

```bash let mut bcs = bcs::new(x"00"); let is_some = bcs.peel_bool();

assert!(is_some == false);

let mut bcs = bcs::new(x"0101"); let is_some = bcs.peel_bool(); let value = bcs.peel_u8();

assert!(is_some == true); assert!(value == 1); ```

If you need to decode an option of a custom type, use the method in the code snippet above.

The most common scenarios, bcs module provides a basic set of functions for decoding Option's:

Structs are decoded field by field, and there is no standard function to automatically decode bytes into a Move struct, and it would have been a violation of the Move's type system. Instead, you need to decode each field manually.

```bash // some bytes... let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

let (age, is_active, name) = ( bcs.peel_u8(), bcs.peel_bool(), bcs.peel_vec_u8().to_string() );

let user = User { age, is_active, name }; ```

Binary Canonical Serialization is an efficient binary format for structured data, ensuring consistent serialization across platforms. The Sui Framework provides comprehensive tools for working with BCS, allowing extensive functionality through built-in functions.

## Using BCS

The Sui Framework includes the sui::bcs module for encoding and decoding data. Encoding functions are native to the VM, and decoding functions are implemented in Move.

To encode data, use the bcs::to_bytes function, which converts data references into byte vectors. This function supports encoding any types, including structs.

```
bash // File: move-stdlib/sources/bcs.move public native fun to_bytes<T>(t: &T): vector<u8>;
```

The following example shows how to encode a struct using BCS. The to_bytes function can take any value and encode it as a vector of bytes.

```bash use sui::bcs;

// 0x01 - a single byte with value 1 (or 0 for false) let bool_bytes = bcs::to_bytes(&true); // 0x2a - just a single byte let u8_bytes = bcs::to_bytes(&42u8); // 0x2a00000000000000 - 8 bytes let u64_bytes = bcs::to_bytes(&42u64); // address is a fixed sequence of 32 bytes // 0x0000000000000000000000000000000000000000000000000000000000000002 let addr = bcs::to_bytes(&@sui); ```

Structs encode similarly to simple types. Here is how to encode a struct using BCS:

```bash let data = CustomData { num: 42, string: b"hello, world!".to_string(), value: true };
```

let struct_bytes = bcs::to_bytes(&data);

let mut custom_bytes = vector[]; custom_bytes.append(bcs::to_bytes(&42u8)); custom_bytes.append(bcs::to_bytes(&b"hello, world!".to_string())); custom_bytes.append(bcs::to_bytes(&true));

// struct is just a sequence of fields, so the bytes should be the same! assert!(&struct_bytes == &custom_bytes); ```

Because BCS does not self-describe and Move is statically typed, decoding requires prior knowledge of the data type. The sui::bcs module provides various functions to assist with this process.

BCS is implemented as a wrapper in Move. The decoder takes the bytes by value, and then allows the caller to peel off the data by calling different decoding functions, prefixed with peel_* . The data is split off the bytes, and the remainder bytes are kept in the wrapper until the into_remainder_bytes function is called.

```bash use sui::bcs;

// BCS instance should always be declared as mutable let mut bcs = bcs::new(x"0100000000000000000");

// Same bytes can be read differently, for example: Option let value: Option = bcs.peel_option_u64();

assert!(value.is_some()); assert!(value.borrow() == &0);

let remainder = bcs.into_remainder_bytes();

assert!(remainder.length() == 0); ```

There is a common practice to use multiple variables in a single let statement during decoding. It makes code a little bit more readable and helps to avoid unnecessary copying of the data.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// mind the order!!! // handy way to peel multiple values let (bool_value, u8_value, u64_value) = ( bcs.peel_bool(), bcs.peel_u8(), bcs.peel_u64() ); ```

While most of the primitive types have a dedicated decoding function, vectors need special handling, which depends on the type of the elements. For vectors, first you need to decode the length of the vector, and then decode each element in a loop.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// bcs.peel_vec_length() peels the length of the vector :) let mut len = bcs.peel_vec_length(); let mut vec = vector[];

// then iterate depending on the data type while (len > 0) { vec.push_back(bcs.peel_u64()); // or any other type len = len - 1; };

assert!(vec.length() == 1); ```

For most common scenarios, bcs module provides a basic set of functions for decoding vectors:

Option is represented as a vector of either 0 or 1 element. To read an option, you would treat it like a vector and check its length (first byte - either 1 or 0).

```bash let mut bcs = bcs::new(x"00"); let is_some = bcs.peel_bool();

assert!(is_some == false);

let mut bcs = bcs::new(x"0101"); let is_some = bcs.peel_bool(); let value = bcs.peel_u8();

assert!(is_some == true); assert!(value == 1); ```

If you need to decode an option of a custom type, use the method in the code snippet above.

The most common scenarios, bcs module provides a basic set of functions for decoding Option's:

Structs are decoded field by field, and there is no standard function to automatically decode bytes into a Move struct, and it would have been a violation of the Move's type system. Instead, you need to decode each field manually.

```bash // some bytes... let mut bcs = bcs::new(x"0101010F0000000000F00000000000");
```

let (age, is_active, name) = (bcs.peel_u8(), bcs.peel_bool(), bcs.peel_vec_u8().to_string());

let user = User { age, is_active, name }; ```

Binary Canonical Serialization is an efficient binary format for structured data, ensuring consistent serialization across platforms. The Sui Framework provides comprehensive tools for working with BCS, allowing extensive functionality through built-in functions.

Encoding

To encode data, use the bcs::to_bytes function, which converts data references into byte vectors. This function supports encoding any types, including structs.

```
bash // File: move-stdlib/sources/bcs.move public native fun to_bytes<T>(t: &T): vector<u8>;
```

The following example shows how to encode a struct using BCS. The to_bytes function can take any value and encode it as a vector of bytes.

```bash use sui::bcs;

// 0x01 - a single byte with value 1 (or 0 for false) let bool_bytes = bcs::to_bytes(&true); // 0x2a - just a single byte let u8_bytes = bcs::to_bytes(&42u8); // 0x2a00000000000000 - 8 bytes let u64_bytes = bcs::to_bytes(&42u64); // address is a fixed sequence of 32 bytes // 0x0000000000000000000000000000000000000000000000000000000000000002 let addr = bcs::to_bytes(&@sui); ```

Structs encode similarly to simple types. Here is how to encode a struct using BCS:

```bash let data = CustomData { num: 42, string: b"hello, world!".to_string(), value: true };

let struct_bytes = bcs::to_bytes(&data);

let mut custom_bytes = vector[]; custom_bytes.append(bcs::to_bytes(&42u8)); custom_bytes.append(bcs::to_bytes(&b"hello, world!".to_string())); custom_bytes.append(bcs::to_bytes(&true));

// struct is just a sequence of fields, so the bytes should be the same! assert!(&struct_bytes == &custom_bytes); ```

Because BCS does not self-describe and Move is statically typed, decoding requires prior knowledge of the data type. The sui::bcs module provides various functions to assist with this process.

BCS is implemented as a wrapper in Move. The decoder takes the bytes by value, and then allows the caller to peel off the data by calling different decoding functions, prefixed with peel_* . The data is split off the bytes, and the remainder bytes are kept in the wrapper until the into_remainder_bytes function is called.

```bash use sui::bcs;

// BCS instance should always be declared as mutable let mut bcs = bcs::new(x"0100000000000000");

// Same bytes can be read differently, for example: Option let value: Option = bcs.peel_option_u64();

assert!(value.is_some()); assert!(value.borrow() == &0);

let remainder = bcs.into_remainder_bytes();

assert!(remainder.length() == 0); ```

There is a common practice to use multiple variables in a single let statement during decoding. It makes code a little bit more readable and helps to avoid unnecessary copying of the data.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// mind the order!!! // handy way to peel multiple values let (bool_value, u8_value, u64_value) = ( bcs.peel_bool(), bcs.peel_u8(), bcs.peel_u64() ); ```

While most of the primitive types have a dedicated decoding function, vectors need special handling, which depends on the type of the elements. For vectors, first you need to decode the length of the vector, and then decode each element in a loop.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// bcs.peel_vec_length() peels the length of the vector :) let mut len = bcs.peel_vec_length(); let mut vec = vector[];

// then iterate depending on the data type while (len > 0) { vec.push_back(bcs.peel_u64()); // or any other type len = len - 1; };

assert!(vec.length() == 1); ```

For most common scenarios, bcs module provides a basic set of functions for decoding vectors:

Option is represented as a vector of either 0 or 1 element. To read an option, you would treat it like a vector and check its length (first byte - either 1 or 0).

```bash let mut bcs = bcs::new(x"00"); let is_some = bcs.peel_bool();

assert!(is_some == false);

let mut bcs = bcs::new(x"0101"); let is_some = bcs.peel_bool(); let value = bcs.peel_u8();

assert!(is_some == true); assert!(value == 1); ```

If you need to decode an option of a custom type, use the method in the code snippet above.

The most common scenarios, bcs module provides a basic set of functions for decoding Option's:

Structs are decoded field by field, and there is no standard function to automatically decode bytes into a Move struct, and it would have been a violation of the Move's type system. Instead, you need to decode each field manually.

```bash // some bytes... let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

let (age, is_active, name) = ( bcs.peel_u8(), bcs.peel_bool(), bcs.peel_vec_u8().to_string() );

let user = User { age, is_active, name }; ```

Binary Canonical Serialization is an efficient binary format for structured data, ensuring consistent serialization across platforms. The Sui Framework provides comprehensive tools for working with BCS, allowing extensive functionality through built-in functions.

## Decoding

Because BCS does not self-describe and Move is statically typed, decoding requires prior knowledge of the data type. The sui::bcs module provides various functions to assist with this process.

BCS is implemented as a wrapper in Move. The decoder takes the bytes by value, and then allows the caller to peel off the data by calling different decoding functions, prefixed with peel_* . The data is split off the bytes, and the remainder bytes are kept in the wrapper until the into_remainder_bytes function is called.

```bash use sui::bcs;

// BCS instance should always be declared as mutable let mut bcs = bcs::new(x"0100000000000000000");

// Same bytes can be read differently, for example: Option let value: Option = bcs.peel_option_u64();

assert!(value.is_some()); assert!(value.borrow() == &0);

let remainder = bcs.into_remainder_bytes();

assert!(remainder.length() == 0); ```

There is a common practice to use multiple variables in a single let statement during decoding. It makes code a little bit more readable and helps to avoid unnecessary copying of the data.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// mind the order!!! // handy way to peel multiple values let (bool_value, u8_value, u64_value) = ( bcs.peel_bool(), bcs.peel_u8(), bcs.peel_u64() ); ```
```

While most of the primitive types have a dedicated decoding function, vectors need special handling, which depends on the type of the elements. For vectors, first you need to decode the length of the vector, and then decode each element in a loop.

```bash let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// bcs.peel_vec_length() peels the length of the vector :) let mut len = bcs.peel_vec_length(); let mut vec = vector[];

// then iterate depending on the data type while (len > 0) { vec.push_back(bcs.peel_u64()); // or any other type len = len - 1; };

assert!(vec.length() == 1); ```

For most common scenarios, bcs module provides a basic set of functions for decoding vectors:

Option is represented as a vector of either 0 or 1 element. To read an option, you would treat it like a vector and check its length (first byte - either 1 or 0).

```bash let mut bcs = bcs::new(x"00"); let is_some = bcs.peel_bool();

assert!(is_some == false);

let mut bcs = bcs::new(x"0101"); let is_some = bcs.peel_bool(); let value = bcs.peel_u8();

assert!(is_some == true); assert!(value == 1); ```

If you need to decode an option of a custom type, use the method in the code snippet above.

The most common scenarios, bcs module provides a basic set of functions for decoding Option's:

Structs are decoded field by field, and there is no standard function to automatically decode bytes into a Move struct, and it would have been a violation of the Move's type system. Instead, you need to decode each field manually.

```bash // some bytes... let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

let (age, is_active, name) = ( bcs.peel_u8(), bcs.peel_bool(), bcs.peel_vec_u8().to_string() );

let user = User { age, is_active, name }; ```

Binary Canonical Serialization is an efficient binary format for structured data, ensuring consistent serialization across platforms. The Sui Framework provides comprehensive tools for working with BCS, allowing extensive functionality through built-in functions.

## Summary

Binary Canonical Serialization is an efficient binary format for structured data, ensuring consistent serialization across platforms. The Sui Framework provides comprehensive tools for working with BCS, allowing extensive functionality through built-in functions.
```