IOTA

Streams Specification

Rev: 1.0 A

Revision History

Revision		Date	Notes
1.0	A	3/12/2020	Initial Release

Table of Contents

Introduction

In this current globalised world, an immense amount of data gets transmitted by millions of devices. Although data can vary wildly in size and structure, a single requirement is valid across all. A way to transmit the data. This can be achieved in numerous ways, although this will thus also result in various different ways to process such. This results in the need of a standardised framework for receiving and sending data to a transportation medium, in order to make it easier for everyone to work with their data.

This document outlines the specifications to our proposal for such a framework, and includes a protocol implementing this framework for securely sending and receiving data over the Tangle.

Scope

The requirements of creating such a framework are vast. That is why we are limiting ourselves to the framework and a protocol. These topics will not include any specifics about the implementation of software that will make use of this. Granular access control is managed at the application layer and can include other frameworks such as IOTA Access for fully embedded Access control and policy management.

The 2 topics of focus in this specification are as follows:

- **Streams** - Defines the message preparation, transportation, and spongos layering. This is the baseline universal framework to be followed.

- **Channels Protocol** - Defines the reference design protocol based on IOTA Streams, which defines the packetization structure, message typing, and cryptographic processes used in development of the Streams implementation. This is also where the current Publish/Subscribe architecture is being controlled.

Users are encouraged to expand on the protocol we propose, and use it as a part of their software. It is not intended as a stand alone, all-encompassing software suite.

Glossary

Table 1-1 : Glossary of Terms

Author	User, creator of a channel, channel owner
Subscriber	User of a channel
Publisher	User role, can publish authenticated messages in a channel, can restrict access to (encrypt) messages for a set of allowed Subscribers. Author always has a Publisher role.
Participant	User of a channel, may be Subscriber or Publisher
Allowed recipient	A Participant listed as allowed to access a certain message (or branch).
Transport	Transport layer abstraction used to deliver messages. Usually, it's Tangle or a transport protocol such as TCP.
Link	Cryptographic mechanism allowing to bind messages.
Message tree	Graph with messages as nodes and links as edges.
Branch	Subtree in message tree.
Chain	Branch which is also a list (ie. each node except for the last one has exactly one child).
Channel	A logical collection of linked messages.
Channel address	A unique identifier for a Channel
Message	A cryptographically processed piece of data consisting of a Header and a Content. Content depends on message content type.
Message ID	Message identifier within a channel.
Message address	Channel address together with message ID.
PRNG	Pseudo-random number generator.

IOTA Streams

Overview

Streams is a framework for developing cryptographic protocols. The framework provides a toolset for structuring and transforming data for application-specific purposes, to be communicated over any transportation layer.

Features included in the framework are:
- Sponge-based automaton for message processing, data encryption and authentication
- Ed25519 signature scheme ([RFC8032](#)) and X25519 key exchange ([RFC7748](#))
- Pseudo-random generator for secure key generation
- Data description and modification language (DDML)

In order to define protocol the following steps are required:
1. define application purpose, protocol goals, user roles;
2. define protocol steps, define messages with DDML, implement message processing;
3. select transport layer protocol;
4. wrap transport and protocol logic into API;
5. use the API in an application.

The Streams framework focuses mainly on step 2.

The main tool the Streams framework provides is DDML. It is a domain specific language and is able to define messages, data fields and rules for cryptographic processing. While there are some great existing protocols and languages, there are none that define both **data representation and cryptographic processing**. DDML can be *interpreted* in two main ways: payload data can be *wrapped* into a binary message and a binary message can be *unwrapped* to extract payload data. DDML supports Spongos automaton operations, signature and key exchange operations, and can be extended with custom cryptographic primitives. The use of Spongos automaton allows embedded/IoT devices to process a packet in a single pass.

The Streams framework is intended for wide use across different types of systems and applications. With such wide support it needs to target different transport media such as the Tangle and TCP. The Tangle can be described as a bag of unordered asynchronous messages. In order for the Tangle to be usable as a transport medium message it needs to be extended with meta-information for packetization and sequencing. Packetization serves as a transport abstraction layer for Channels protocol. Sequencing allows Channels protocol messages to be ordered in a suitable and convenient way.

Cryptography

Cryptographic capabilities include symmetric cryptography via spongos automaton as well as Ed25519 signature scheme and X25519 key exchange. Spongos automaton provides data integrity and confidentiality and can be used to generate pseudo-random data. Spongos automaton is a sponge construction and uses Keccak-f[1600] as permutation. The choice of primitives is such as to provide 128 bits of classical security.

Spongos Automaton

Spongos automaton is a symmetric cryptography scheme based on sponge construction. The automaton acts as a finite state machine, ie. it maintains an internal state and supports a number of operations. The state is modified by a pseudo random permutation. The spongos state consists of an inner secret part and an outer part available for the use by spongos operations. The inner part of the state guarantees cryptographic security and must be kept in secret. The size of the inner part is called capacity. The size of the outer part is called rate and determines the size of input/output data block. In this specification the permutation is fixed to [Keccak-f[1600]](#) with the state size of 1600 bits, 1344 bit rate and 256 bit capacity.

The spongos operations take byte string as input or no input at all, modify internal state and produce byte string as output or no output at all. The operations are constructed to provide data confidentiality, confidential and additional data integrity and authenticity, and generate pseudo random data. The spongos automaton functions by calling several operations in succession and can be used to implement interfaces of a hash function, authenticated encryption, or PRNG.

Notation

⊥	Empty value or argument
u <- a	Assign a value `a` to a variable `u`
$\{0,1\}^n$	The set of all binary words of length `n`
$\{0,1\}^*$	The set of all binary words of finite length
α^n	The concatenation of α with itself `n` times
|X|	The length of a binary word `X`
X[i]	The `i`-th bit of a binary word `X`
a ∥ b	The concatenation of two binary words `a` and `b`
F	Spongos permutation, currently Keccak-f[1600] is used
R	Rate of the spongos state, size in bits of data that spongos automaton is capable of processing per each permutation call, `R = 1344`
N	Width (size in bits) of the spongos state, `N = 256 + R = 1600`

Interface

Spongos automaton maintains an internal state: a permutation buffer `s` and the current offset `pos` within it. The buffer should be kept in secret as it contains the required data to process and possibly mask the next message. Spongos internal state is passed implicitly and is available to the automaton interface functions.

Init

Spongos automaton state initialization, it's the first command executed by the automaton.

```
Init
Input : ⊥
Output: ⊥
Steps :
   1. S <- 0^N
   2. pos <- 0
```

Commit

Commit changes to the buffer. The command is usually called between absorbing and squeezing commands.

```
Commit
Input : ⊥
Output: ⊥
Steps :
   1. if pos ≠ 0:
         a. S <- F(S)
         b. pos <- 0
```

Update

Internal command that commits current rate part into the buffer.

```
Update
Input : ⊥
Output: ⊥
Steps :
   1. if pos = R:
         a. Commit()
```

Absorb

Process input data.

```
Absorb
Input : X∈{0,1}*
Output: ⊥
Steps :
   1. for i=0,...,|X|-1:
         a. S[pos] <- X[i]
         b. pos <- pos + 1
         c. Update()
```

Squeeze

Produce output data.

```
Squeeze
Input : n∈ℕ
Output: Y∈{0,1}ⁿ
Steps :
   1. Y <- 0ⁿ
   2. for i=0,...,n-1:
        a. Y[i] <- S[pos]
        b. pos <- pos + 1
        c. Update()
   3. return Y
```

Encrypt

Encrypt plaintext. Key must be absorbed and committed before.

```
Encrypt
Input : X∈{0,1}*
Output: Y∈{0,1}ⁿ,n=|X|
Steps :
   1. Y <- 0ⁿ
   2. for i=0,...,|X|-1:
        a. Y[i] <- X[i] ⊕ S[pos]
        b. S[pos] <- Y[i]
        c. pos <- pos + 1
        d. Update()
   3. return Y
```

Decrypt

Decrypt ciphertext. Key must be absorbed and committed before.

```
Decrypt
Input : Y∈{0,1}*
Output: X∈{0,1}ⁿ,n=|Y|
Steps :
   1. X <- 0ⁿ
   2. for i=0,...,|X|-1:
        a. X[i] <- Y[i] ⊕ S[pos]
        b. S[pos] <- Y[i]
        c. pos <- pos + 1
        d. Update()
   3. return X
```

Fork

Duplicate spongos automaton.

```
Fork
Input : ⊥
Output: S'
Steps :
    1. S' <- S
    2. return S'
```

Join

Join two spongos automatons.

```
Join
Input : S'
Output: Y∈{0,1}ⁿ,n=|X|
Steps :
    3. S'.Commit()
    4. T <- S'.Squeeze(R)
    5. Absorb(T)
    6. Commit()
```

PRNG

Pseudo-random number generator is constructed using a spongos automaton.

Init

Initializes PRNG with a secret key.

```
Init
Input : K∈{0,1}²⁵⁶
Output: ⊥
Steps :
    1. S'.Commit()
    2. T <- S'.Squeeze(R)
    3. Absorb(T)
    4. Commit()
```

Gen

Generate pseudo-random numbers.

```
Gen
Input : N∈{0,1}* (nonce), n∈ℕ
Output: Y∈{0,1}ⁿ
Steps :
   1. S.Init()
   2. S.Absorb(K ∥ d ∥ ⟨|N|⟩ ∥ N ∥ ⟨n⟩)
   3. S.Commit()
   4. Y <- S.Squeeze(n)
   5. return Y
```

Ed25519

Ed25519 is a signature scheme in EdDSA family ([RFC8032](#)) defined over the Curve25519 in Edwards form. The scheme consists of three algorithms: key generation, hash signing and signature verification. Private key is a 256-bit random number. Public key is a 256-bit number deterministically calculated from a private key.

Gen

Generate key pair.

```
Gen
Input :
Output: sk∈{0,1}²⁵⁶
Return: Ed25519ctx(sk; h)
```

Sign

Sign hash.

```
Sign
Input : sk∈{0,1}²⁵⁶, h∈{0,1}⁵¹²
Output: sig∈{0,1}⁵¹²
Return: Ed25519ctx(sk; h)
```

Verify

Verify signature.

```
Verify
Input : pk∈{0,1}²⁵⁶, h∈{0,1}⁵¹², sig∈{0,1}⁵¹²
Output: {0,1}
Steps :
   5. S'.Commit()
   6. T <- S'.Squeeze(R)
   7. Absorb(T)
   8. Commit()
```

X25519

X25519 is a Diffie-Hellman key exchange algorithm ([RFC7748](#)) defined over the Curve25519 in Montgomery form.

Gen

Generate key pair.

```
Gen
Input :
Output: sk∈{0,1}^256
Return: Ed25519ctx(sk; h)
```

X25519

Produce a shared secret.

```
X25519
Input :
Output: sk∈{0,1}^256
Return: Ed25519ctx(sk; h)
```

Streams DDML

Streams Data Description and Modification Language (DDML) is a message description language that has been designed with both data organization and cryptographic functionality. A message is composed of individual fields and a field is defined by its type and command. A *field type* describes how data should be represented (i.e. encoding), while a *field command* describes how data should be processed.

Messages are composed using a combination of cryptographic field commands alongside predefined field types to create and parse data uniformly within the application environment. These functions are referred to as *"Wrap"*(create) and *"Unwrap"*(parse) respectively. Additionally, for buffer sizing purposes, there is a *"SizeOf"* set of functions for determining the required size of buffer for a message being prepared. Each of these sets of functions forms a context using the same function names.

A *"Wrap"* function takes a Spongos state, application context (keys, RNGs, auxiliary data) and input data (payload) and produces a binary encoded message, as well as the resulting Spongos state. Conversely, an "Unwrap" takes a Spongos state, application context (keys, aux data) and binary encoded message to produce output data (payload) and the resulting Spongos state.

The resulting Spongos state after a *"Wrap"* and *"Unwrap"* operation will only be equal if the same initial Spongos state and corresponding keys were used to process a message on each end. These Spongos states are required by all participants in order to be synchronised.

Messages built from malformed sponges states are not processed, as the message identifier's will not match the defined pattern, and the states may not contain the correct session keys/nonces necessary for participant processing.

Syntax

Types

`u8`	Unsigned 8-bit integer
`u16`	Unsigned 16-bit integer
`u32`	Unsigned 32-bit integer
`u64`	Unsigned 64-bit integer
`uint`	Unsigned variable length integer
`T[n]`	Fixed-sized array of values of type `T` of length `n`

Modifiers

`oneof`	Construct a choice type
`repeated`	Construct variable-length array
`external`	Field is stored out-of message, in the DDML program context

Commands

`absorb`	Empty argument
`squeeze`	Assign a value `a` to a variable `u`
`mask`	The set of all binary words of length `n`
`commit`	The set of all binary words of finite length
`fork`	Fork spongos automaton, continue processing the associated field with the forked automaton
`join`	

Encoding Rules

$\perp$	Empty argument
`u <- a`	Assign a value `a` to a variable `u`
`{0,1}`^{n}	The set of all binary words of length `n`
`{0,1}*`	The set of all binary words of finite length

Commands Summary

Table 2-1 : Glossary of Terms

Command	Wrap Context	Unwrap Context	SizeOf Context*
absorb	Spongos absorb input bytes	Spongos absorb extracted output stream bytes	+= buf_len
absorb_external	Spongos absorb internal input bytes of defined external type	Spongos absorb extracted output stream bytes for defined external type	+=internal_ buf_len
commit	Spongos commit	Spongos commit	Does nothing
dump	Dump input stream debugger data	Dump output stream debugger data	Print size of dump
ed25519	Input stream absorb ed25519 hash signature	Verify ed25519 Signature	+= ed25519 Signature Size
fork	Spongos fork, stores copy in self, returns self	Spongos fork, stores copy in self, returns self	Return copy of context
guard	Ensure input condition is met	Ensure input condition is met	N/A
join	Fetch linked spongos state, append to beginning of current sponges state	Fetch linked spongos state, append to beginning of current sponges state	+= self id size
mask	Spongos encrypt input bytes	Spongos decrypt output bytes	+= masked_bytes_len
repeated	Perform input function a given number of times	Perform input function a given number of times	Perform input function a given number of times
skip	Input stream absorb bytes for size indicator (used to define length of vector/set)	Output stream extract bytes for size indicator (used to define length of vector/set)	Increase size of input context by input size
squeeze	Spongos squeeze state to Mac size hash	Ensure output stream is Mac sized	+= Mac size
x25519	Spongos absorb x25519 public_key as well as an encrypted payload associated with this key	Extract x25519 public key and decrypt internal payload	+= (Public_key_size + key_len)

* = Some operations within the SizeOf context do nothing, as no additional data will be absorbed in that step. They are maintained for continuity between contexts.

Message Preparation

A "Message" within the context of the Streams framework denotes a uniformly prepared structure of data that can assume any number of responsibilities from a higher level. For the framework, a message itself is constructed using a pre-designated Header space, and an accompanying content structure for placing the senders data into. The Header Descriptor Frame is the frame at the beginning of a message used for indicating the version of streams that is being used, a message identifier for the message preceding the current one, and a type indicator for the message that is being sent. During message generation, the header is always created first, and placed before the Payload Carrying Frames being sent. The content section itself can assume many forms depending on the operations intended to be conducted on the application layer.

```
message Message {
    Header header;
    Content content;
    commit;
}

message Header {
    absorb u8 version;
    absorb external link address;
    absorb u8 content_type;
}

message Content; // app-specific

// Example message
message MyAppContent {
    absorb u8 nonce[16];
    absorb u8 key[32];
    commit;
    mask bytes payload;
    commit;
    ed25519 u8 sig[64];
}
```

Message Transportation

The Streams framework is transportation agnostic. It only requires a variable amount of data to be read or sent at a location. Data will be read according to the specifications of the version this data was created with. The version can be read from the header description frame.

Interface for a transport layer
```
Send_Message(Link, Message, Send_Options)
Receive_Message(Link, Recv_Options): Message
```

Packetization and Framing

Message payloads must be broken down into even portions to be packetized. All of the metrics for the derived packets, along with application level information, are stored in the Header Descriptor Frame (HDF) which is the first frame sent/received when processing a message. The Payload Carrying Frames (PCFs) are the subsequently linked messages containing payload data along with pertinent custom application data.

Both HDF and PCF currently have unused bits marked in black. These areas are kept in the case of a future version requiring more space. When a bit is unused it should be set to zero.

Header Descriptor Frame (HDF)

The Header Descriptor Frame of a message contains important application processing information regarding the proceeding payload frames. This includes the protocol version, message type and encoding type of the content that is being packetized along with custom application data needed for message processing. The Uniform Payload Length has a minimum requirement to be 1 byte, and a maximum available space of 1024 bytes per fragment to coincide with available space in an IOTA transaction. In other transportation schemes (especially within IoT related frameworks) it is possible for this payload length to be reduced for processing on both ends.

Among the remaining contents of the HDF are the frame type identifier, which indicates that this frame is an HDF, as well as a counter for the number of proceeding frames. Frame count maxes out at 2^{21}, meaning a message with Uniform Payload length 1024 bytes can be used to send a maximum message size of 2Gb. Any dataset larger than this will need to be fragmented into separate messages.
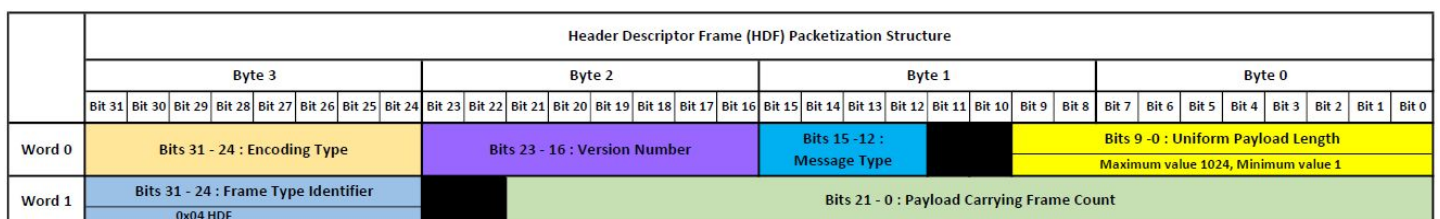
The Encoding Type Field is used to indicate the type of encoding the message is being transmitted in.

Figure 1: HDF Required Fields

Encoding Type Field

Table 2-2 : Encoding Type Values

Register Value	Message Type
0x00	ASCII
0x01	Raw Binary
0x02	Unicode
Other Values	User Defined Encoding Types

Version Number Field

The Version Number Field is used to define which version of IOTA Streams has been used to publish the message. This register is also used to derive what type of receiving end processing will be needed based on the differences between IOTA Streams versions.

Message Type Field

The Message Type Field is used to indicate one of the many message types in a streams implementation. The current register values used are shown below.

Table 2-3 : Message Type Values

Register Value	Message Type
0x0	ANNOUNCE
0x1	KEYLOAD
0x2	SIGNED_PACKET
0x3	TAGGED_PACKET
0x4	SUBSCRIBE
0x5	UNSUBSCRIBE

Uniform Payload Length Field

The Payload Length Field contains the length each Payload Carrying Frame carries in bytes. The maximum value is 1024 bytes and the minimum value is 1 byte.

Frame Type Identifier Field

The Frame Type Identifier Field is used to indicate what type of frame is being processed.

Table 2-4 : Frame Type Identifier Values

Register Value	Message Type
0x04	Header Descriptor Frame
0x05	First Payload Carrying Frame
0x0C	Intermediary Payload Carry Frame
0x0E	Final Payload Carrying Frame
Other Values	User Defined Frame Types

Payload Carrying Frame Count Field

The Payload Carrying Frame Count Field contains the total number of payload carrying frames expected in the message.

Payload Carrying Frame (PCF)

A Payload Carrying frame is the part of a message that contains application specific data.The frame type identifier and current PCF number must be Included in a PCF. The PCF number will allow for ordering and organising of message fragments, while the frame type identifier will indicate if the active PCF is an initial, intermediate or final payload frame.

The payload fragment is placed into the Payload Data Bytes field, conforming to the size constraints issued in the HDF Uniform Payload Carrying Length. This means that the Payload does not have to be 256 words long and can be truncated for resource preservation.

Figure 2: PCF Required Fields

Frame Type Identifier Field

The Frame Type Identifier Field is used to signal what type of frame is being processed. For the Payload Carrying Frames there are three possible values used to indicate initial, intermediary, and final Payload Carrying Frames.

Table 2-5 : Frame Type Identifier Values

Field Value	Message Type
0x05	First Payload Carrying Frame
0x0C	Intermediary Payload Carry Frame
0x0E	Final Payload Carrying Frame

Payload Carrying Frame Number Field

The Payload Carrying Frame Number Field contains the current Payload Carrying Frame number that is being processed.

Payload Data Field

The Payload Data Field is the field in which the actual payload data is placed little endian, in 32-bit words.

Transport over the Tangle

Binary-to-Trinary Conversion

Transaction format uses trinary encoding. Transport over the Tangle uses transaction format as a unit of messaging. Streams messages are binary-based. This section specifies the way to represent a binary message in trits.

> **Possible Binary-to-Trinary conversions include:**
>
> ▸ 4 bits (0..15) to 1 tryte (0..26), ie. 1 byte to 2 trytes
> ▸ 16 bit to 11 trits
> ▸ Integer conversion: n-bit integer (0..2^n-1) to [log_3(2^n)]+1 trits

Channels

The Channels protocol is a protocol designed to operate within the Streams framework. Channels can be used as the replacement for the previously created MAM, which controls and structures data, but is not limited to this feature. It can be easily expanded upon.

Channels core functionality is "providing a secure off-line messaging implementation".

Overview

Channels is a protocol implementation that uses the IOTA Protocol as a transportation layer. Its core functionality is achieved through the following features:

▸ Maintains Streams state through an internal link store mechanism
▸ Numerous predefined message types (i.e. Signed Packets, Keyloads, etc.)
▸ Decentralised transportation and storage through the usage of the Tangle
▸ Provides message types for managing cryptographic access control to branches of data
▸ Uses a pub/sub model with key sharing for access management.

While the Channels protocol utilises the Tangle for the integrity of the data being transmitted, the application layer itself provides authenticity and protection through direct association of data with the source, and key management solutions for assigning read/write privileges to any particular branch. All publishing entities generate an Ed25519 and X25519 key pairing to be used for signing and encryption respectively, and these keys are exchanged via subscription messages. Once subscribed, a participant can then be authorised by the channel author to participate in or read from any number of branches.

Branch access can only be given and not retroactively removed, so in the event that unsubscription is required, a new branch must be generated, or a new access policy must be sent within the branch,

informing participants that the unsubscribed user is no longer participating in the reading and writing of that branch. On the subscribers end, they will no longer be able to process data beyond that branch/policy change.

Messages of various types and purposes are organised to form "trees" of referenced transaction points. These message references in turn act as an organisational and navigational set of tools for traversing data posted to the tangle. The variety in message types provides a KPI structure between parties communicating through the network, enabling a distributed model of data management with a built in degree of discretion for posted data.

Channels Specific Packetization

The Channels Application needed the following custom fields added to the HDF and PCF for message processing:

- ▸ **Message Identifier** of the linked message preceding the current message being generated.
- ▸ **Sequence Number** state of the Publisher for the current message..

Figure 3: Channels Protocol HDF

	Payload Carrying Frame (PCF) Packetization Structure			
	Byte 3	Byte 2	Byte 1	Byte 0
	Bit 31 Bit 30 Bit 29 Bit 28 Bit 27 Bit 26 Bit 25 Bit 24	Bit 23 Bit 22 Bit 21 Bit 20 Bit 19 Bit 18 Bit 17 Bit 16	Bit 15 Bit 14 Bit 13 Bit 12 Bit 11 Bit 10 Bit 9 Bit 8	Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0
Word 0	Bits 31 - 24 : Frame Type Identifier 0x05 Initial PCF (PCF 0) 0x0C Intermediatary PCF 0x0E Final PCF		Bits 21 - 0 : Payload Carrying Frame Number	
Word 1	Payload Data Bytes 0 : 3 *Little Endian with Max Payload size of 1024. Words that are truncated are padded with 0s*			
Word 2	Payload Data Bytes 4 : 7			
...	...			
Word 256	Payload Data Bytes 1020 : 1023			
Word 257	*Previous Message ID Bytes 0 : 3			
Word 258	*Previous Message ID Bytes 4 : 7			
Word 259	*Previous Message ID Bytes 8 : 11			
Word 260	*Sequence Number Bytes 0 : 3			
Word 261	*Sequence Number Bytes 4 : 7			

Figure 4: Channels Protocol PCF

Message Types

In order to create the functionality required for Channels to accomplish its tasks, message types are needed. Each message will handle a specific task in the Channels logic by following a predefined chain of cryptographic instructions. A field inside the HDF will signify what type of message should be parsed.

▸ **ANNOUNCE** - Message generated by Author that creates channel and designates Root Address and Message ID. This message indicates the very beginning of a channel, and will always be the first message of a channel. After absorbing the required information, the participant will verify the signature against the expected public key to ensure the message came from the expected author. Once that is verified, the multi-branch setting is detected and set accordingly in order to parse further messages.

```
DDML
message Announce {
    absorb u8 x25519_pubkey[32];
    absorb u8 flags;
    commit;
    squeeze u8 hash[64];
    ed25519(hash) sig;
}
```

▸ **KEYLOAD** - Message generated by the Publisher. It contains a list of public keys of each permissioned Subscriber in the channel, along with the randomly generated key for chaining. (The nonce) Allowed Recipients are identified either by pre-shared keys or by x25519 public key identifiers. This is used in creating branches.

```
DDML
message Keyload {
    join link msgid;
    absorb u8 nonce[32];
    skip repeated {
        fork;
        mask u8 pk_id[16];
        absorb external u8 pre_shared_key[32];
        commit;
        mask u8 key[32];
    }
    skip repeated {
        fork;
        mask u8 pk_id[16];
        x25519(key) u8 ekey[32];
    }
    absorb external u8 key[32];
    commit;
 }
```

▸ **SIGNED_PACKET** - Message from Publisher signed using the Publisher's private key that is appended to a channel message chain. It contains both plain and masked payloads. The message can only be signed and published by the channel owner. Channel owners must first publish their corresponding public key certificate in either `Announce` or `ChangeKey` message, so the receivers can verify that it came from the owner.

```DDML
message SignedPacket {
    join link msgid;
    absorb uint public_size;
    absorb u8 public_payload[public_size];
    absorb uint masked_size;
    mask u8s masked_payload[masked_size];
    commit;
    squeeze u8 hash[32];
    ed25519(hash) sig;
}
```

▸ **TAGGED_PACKET** - Unsigned message from either Publisher or Subscriber that is appended to a channel message chain.

```DDML
message TaggedPacket {
    join link msgid;
    absorb uint public_size;
    absorb u8 public_payload[public_size];
    absorb uint masked_size;
    mask u8s masked_payload[masked_size];
    commit;
    squeeze u8 mac[32];
}
```

▸ **SUBSCRIBE** - Message from Subscriber indicating a subscription to a channel.

```DDML
message Subscribe {
    join link msgid;
    x25519(key) u8 unsubscribe_key[32];
    commit;
    mask u8 pk[32];
    commit;
    squeeze u8 mac[32];
}
```

▸ **UNSUBSCRIBE** - Message from Subscriber indicated an unsubscription to a channel.

```
DDML
message Unsubscribe {
    join link msgid;
    commit;
    squeeze tryte mac[27];
}
```

▸ **SEQUENCE** - Special message type used as a reference pointer for other messages in a multi branch implementation. Contains the essence necessary to derive the referenced message's identification marker for retrieval.

```
DDML
message Sequence {
    join link msgid;
    absorb u8 ke_pk[32];
    skip u64 seq_num;
    absorb u8 linked_msg_id;
    commit;
    squeeze u8 hash[32];
    ed25519(hash) sig;
}
```

Branching and Sequencing

Messages generated through the Channels protocol can be attached in a variety of customisable "tree" shapes. The customisable nature of these trees can make navigation and key management complicated quickly, especially in a multi publisher environment. Two model configurations exist that all channel shapes are derived from: "Single Branching" and "Multi Branching". Branching refers to the delineation of participant management within a channel. The channel author can place user keys into Keyload messages to indicate the participants that can publish and retrieve messages sent afterwards within that chain or sub-branches. Upon creation of a new channel, a flag indicating which of these two models is to be used must be specified to determine the sequencing model logic that all participants will be configured with.

Single Branch

In a single branch channel, all messages will be incrementally sent to a single branch by all publishers within the channel. Each time a publisher posts a message into the chain, all participants increment the sequence state uniformly. This allows for easy transmission of data between directly connected devices (i.e. a subscription to sensor streams) where all data posted is relevant to all parties involved. When looking for the next message, a message identifier can be generated for each public key using the next anticipated sequence state. If the message is found and verified, that sequence state is updated and the next message searched for.
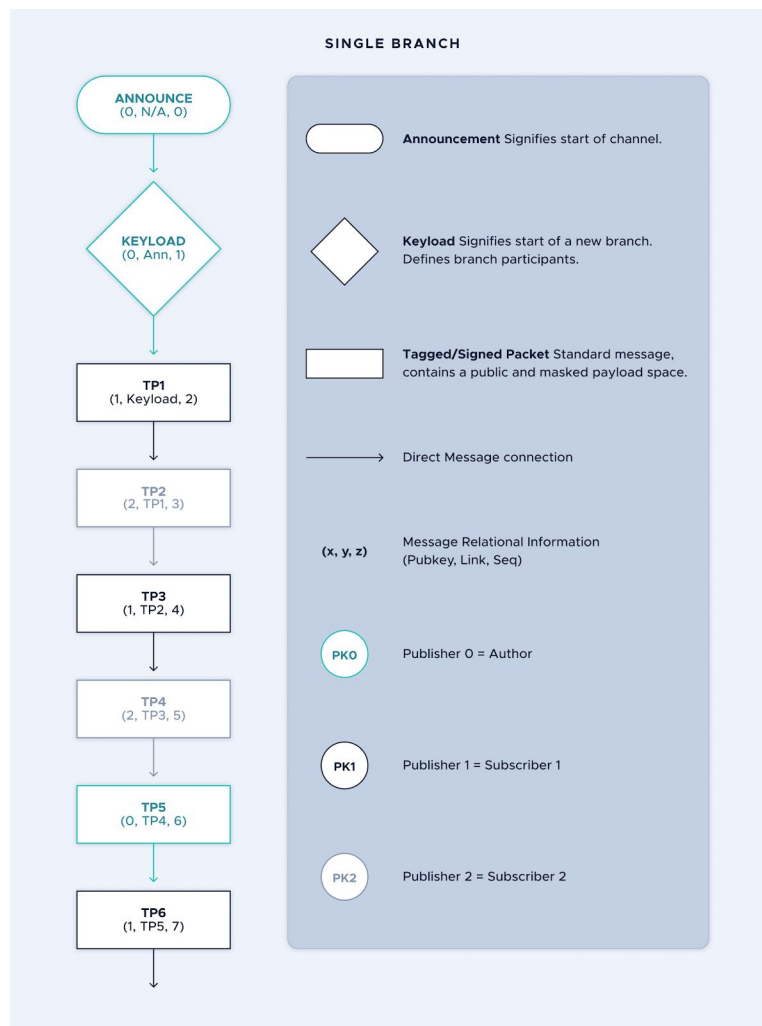
Figure 5: Single Branch Implementation Example

Multi-Branch

In a "Multi-Branch" channel, messages can be placed into a hierarchy of "branches" and "chains" within the message tree. Each branch can contain several chains, attaching to previously existing messages in a variety of orders and configurations. In order to organise these data points, a special sequencing approach is used. Once a new publisher has been established in the channel, the channel author will inform the other participants of the existence of this new publisher, and a new "chain" of messages will be formed for that publisher from the channel announcement message. Each message sent to another branch in the channel will trigger a sequencing reference message to be placed into that publisher's sequencing chain. This allows participants to traverse a known publisher's sequenced messages without needing to actively monitor all active branches in the channel simultaneously. These sequence messages will provide the essence necessary to derive the identifier for the referenced message for retrieval. Provided the participant has read privileges to the branch, they will then be able to process the data contained in the message.
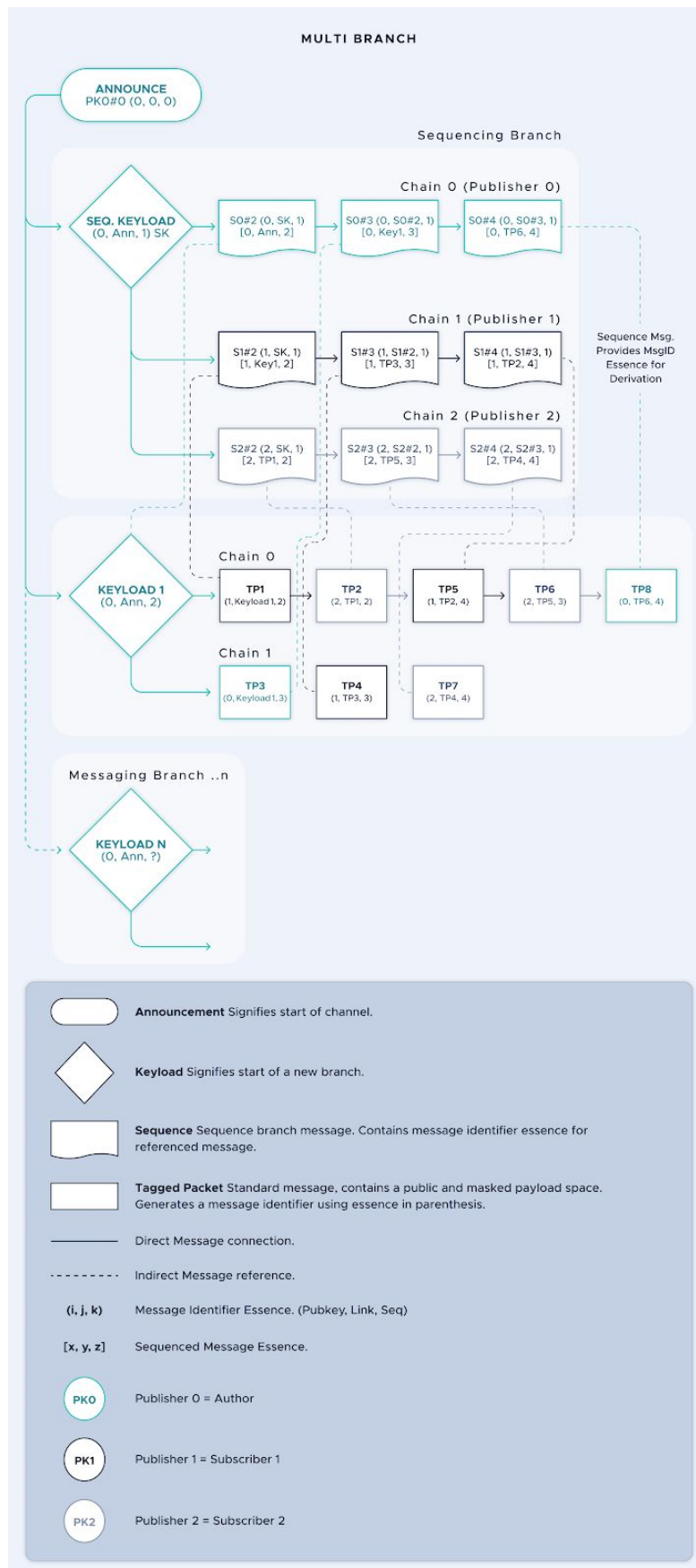
Figure 6: Multi-branch Implementation Example

Transportation & Storage On the Tangle

Messages sent through the Channels protocol packetize into IOTA transactions before being sent to a node for attachment to the Tangle DAG (Directed Acyclic Graph). With respect to message location within the Tangle, there are two reference points to use during attachment and retrieval: The Transaction Address, and the Attachment Tag. In the Channels protocol, these are represented as the Channel Application Instance and Message Identifier respectively. The content/payload of a message is placed into the Signature Message Fragment within the transaction anatomy.
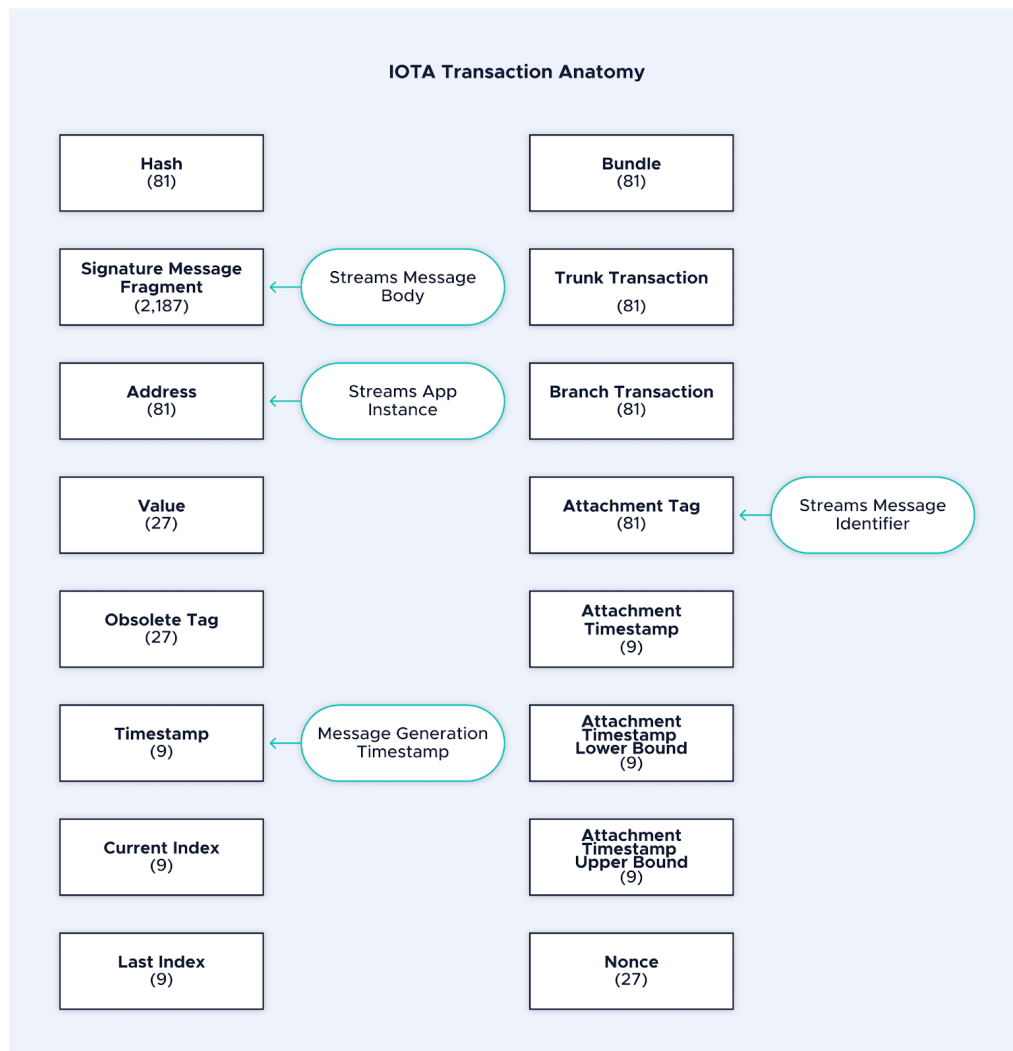
Figure 7: IOTA Transaction Anatomy

Multiple fragments can exist within the same Address/Tag combination. Individual message fragmenting and ordering is handled through IOTA Transaction Bundling Logic, which takes the associated fragments and provides them an index within the context of the bundle for that message identifier. The Channels protocol will parse the message to be sent and determine the number of transactions necessary for message bundling, and will fragment the message into the available space in each transaction. Due to the usage of the Application Instance and Message Identifier's as the Address and Tag of the transactions explicitly, they are not required in the HDF's and PCF's that will be placed into the transactions.

When bundling transactions together, the Channels protocol will first generate the 24 Byte HDF and append it to the beginning of the first transaction. The remaining space in this transaction will be occupied with the initial PCF. If the message is larger than the 1024 Bytes, then the remaining content will be fragmented into further 1024 Byte chunks and placed into evenly sized PCF's within their own transactions. There are 3 uniform types of PCF available to aid in processing: an Initial, Intermediary and Final PCF. The formatting of these PCF's is the same, but the type will indicate the position within a message bundle for processing. The final chunk of a message fragment must be of type Final, even if the message is compact enough to fit within a single transaction. Within an Iota Bundle, only the Final PCF is used, as the bundle handles ordering inherently.
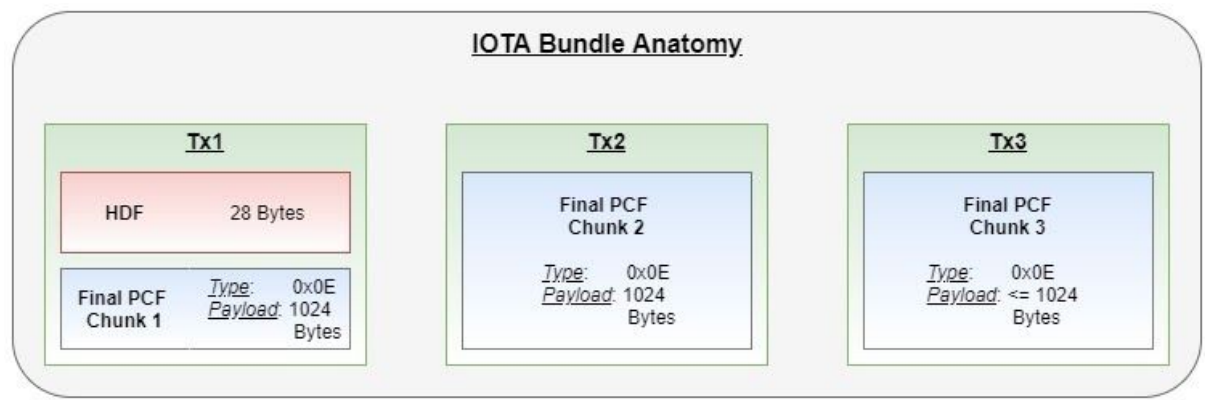

Figure 8: IOTA Bundle Anatomy

For more information

More information about the IOTA Foundation, IOTA protocol and IOTA Tangle can be found on the official website at https://iota.org. To follow the progress on the work that the IOTA Foundation and it's ecosystem are making on the solutions outlined in the document, you can also find more information at the links below.

	Medium	Link	Frequency
	Discord	https://discord.iota.org	All news is announced on Discord
	Twitter	https://twitter.com/iotatoken	All news is announced on Twitter
	Blog	https://blog.iota.org	All news is announced on our blog
	YouTube	https://www.youtube.com/iotafoundation	Announcements, use-cases, tutorials, informational videos
	Newsletter	https://newsletter.iota.org/	A monthly overview and recap

IOTΛ