Indexed Data Transfer Protocol

“The Indexed Data Transfer Protocol (IDTP) introduces a new paradigm for IoT data management by organizing system information as a set of uniquely indexed variables, enabling precise and efficient read/write operations. Unlike traditional transmission-focused protocols, IDTP emphasizes persistence and reliability by maintaining the latest state of each variable and instantly synchronizing updates across all connected devices. This indexed, modular approach ensures real-time communication, minimizes unnecessary data transfer, and provides robust scalability for complex IoT ecosystems.

Protocol Version: v0.4.0-beta

Author: Malison F. Picado S.

Table of Contents

Indexed Data Transfer Protocol (IDTP)

The Indexed Data Transfer Protocol (IDTP) is a protocol based on reading and writing a set of system variables, in which each variable is identified by a unique index. The definition is given as: "A system with a set of variables V, such that: $V = \{V_0, V_1, ..., V_n\}$". V is a set of variables arranged in the form of a matrix of N columns and 1 row, with N starting at 0. Each variable has an assigned data type, which can vary depending on the access mode configured on the server.

Each variable in the set V is associated with an index N, allowing direct and efficient access to the data. Furthermore, each variable can be linked to one or more devices that depend on it, facilitating the management and control of system elements. This indexed and modular structure of PTDI makes it a robust and scalable solution for data transfer in IoT applications. Every time the value of a variable is updated, the new value is sent to each of the devices that depend on that variable; clients receive updates for all variables in the system. All updates are in real time.

Entities

An entity is any device that establishes a connection to the system (broker), whether or not it maintains the connection, and that performs read, write, or modification operations on the set of variables or on an individual variable. An entity can be a microcontroller, a sensor, a web server, or an application; if it is connected to the broker, then it is referred to as an entity.

This protocol distinguishes between two types of entities, which are treated differently by the broker depending on their type. The first type of entity is called a "client." Clients are any connections made to obtain or perform any update to the set of variables; these clients can be web or native applications for monitoring variables. The second type of entity is called a "device." Devices are all connections made to perform only read and write operations on a given number of variables (a subset S

belonging to the set of variables V, $S=\{V_a,...,V_x\}, S\in V$).

Thus, a client is a connection that will receive updates to any variable and can make updates to any variable; a device is a connection that will receive updates to certain variables and send updates to certain variables (declared in the connection request).

$C=\{C_a,C_b,...,C_x\}$, C is the set of clients;

$D=\{D_a,D_b,...,D_x\}$, D is the set of devices;

$D_n(S_u,S_d); S_u=\{V_n,...,V_m\}; S_d=\{V_i,...,V_j\}; S_u,S_d\in V$.

D_n is a device that performs updates to the subset of variables S_u, and depends on (receives updates from) the subset of variables S_d; where S_u and S_d belong to the set of variables V. The subsets S_u and S_d are referred to as parameters. These parameters are declared by a device when making a persistent connection request. Depending on the mode of operation, these parameters may be omitted or processed differently.

Operation Mode

The operation mode is how the broker allows an entity to perform certain operations and process them in a specific manner. There are three modes: strict mode, normal or default mode, and free mode.

Free Mode

Free mode allows any connection without authentication, allows one-time connections, and persistent connections for both devices and clients. Furthermore, it enables any entity to perform all

types of operations without any restrictions, i.e., read and write any variable, declare only dependencies, and override the data type of a variable.

Normal Mode

Normal mode allows one-time connections and requires persistent connections to authenticate in order to perform operations. Device connections are not allowed to change the data type of a variable (only clients using the set type command).

Strict Mode

Strict mode does not allow one-time connections; all connections must authenticate and declare the variables they depend on and the variables they update. If the device does not declare parameters, the connection is rejected; there must be at least one parameter, either an update parameter or a dependency parameter. Devices cannot change the data type when using the UPDATE method (only clients can do so with the SET TYPE command).

Action	Strict Mode	Normal Mode	Free Mode
One-time connections	Prohibited	Allowed	Allowed
Persistent connection	With Authentication	With Authentication	Allowed
Data type overriding	Clients Only	Allowed	Allowed
Updates to any variable. *See Note Below	Clients Only	Allowed	Allowed
Perform variable set expansion operation	Clients Only	Clients Only	Restricted
Parameter definition (updates and dependencies)	Devices Only	Devices Only	Devices Only

Note: Normal mode and free mode allow updates to any variable, both clients and devices, without being declared as a parameter. Strict mode, for devices, only allows updates to variables declared in

parameters: $D_n(S_u, S_d)$; if the device attempts to update the value of a variable that was not previously declared, the update will be rejected; only clients can update the value of any variable in strict mode.

Behavior if Parameter Indexing is Enabled

A device has the option to enable parameter indexing. If the configuration mode is strict, the only difference will be the parameter transmission and reception format, see Parameter Indexing. If the configuration mode is set to normal or free, then the device that chooses to use parameter indexing will be required to perform the operations declared in the parameters. Therefore, the device must declare at least one parameter for the connection to be valid; the device will not be able to perform updates or queries to variables that were not declared in the parameters.

In normal or free mode, the device will behave as if it were in strict mode if the parameter indexing option is enabled for the device; devices that do not choose to enable this feature will be able to freely write and read any variable.

Response Codes

Response codes are a numerical representation of the status of an operation. These codes will be 1 byte long and will precede the request result, indicating its status. The code for any successful operation status is 0x00.

Code	Hex Code	Description
0	0x00	Successful operation
1	0x01	Invalid index
2	0x02	Data type overwrite not allowed
3	0x03	Incomplete payload
4	0x04	Unknown method
5	0x05	Unsupported protocol version

6	0x06	Invalid protocol version
7	0x07	Unknown entity type
8	0x08	Invalid Keep Alive
9	0x09	Connection requires authentication
10	0x0A	Authentication failed
11	0x0B	Connection must have at least one parameter
12	0x0C	One-time connections are not allowed
13	0x0D	Unrecognized data type
14	0x0E	Invalid data type, set to default data type
15	0x0F	Parameter indexing enabled, but no parameters were declared
30	0x1E	Variable could not be added to the array. Expansion limit reached

Data Types

The Indexed Data Transfer Protocol (IDTP) establishes 11 data types and sets the default data type as the 32 bits signed integer (int32, code: 0x07):

Data Type	Code	Hex	Binary	Length Bytes
Boolean	0	0x00	0000	1
8-bits Unsigned Integer(uint8)	1	0x01	0001	1
16-bits Unsigned Integer (uint16)	2	0x02	0010	2
32-bits Unsigned Integer (uint32)	3	0x03	0011	4
64-bits Unsigned Integer (uint64)	4	0x04	0100	8
8-bits Signed Integer (int8)	5	0x05	0101	1
16-bits Signed Integer (int16)	6	0x06	0110	2
32-bits Signed Integer (int32)	7	0x07	0111	4
64-bits Signed Integer (int64)	8	0x08	1000	8
32-bit Float (float32)	9	0x09	1001	4
64-bit Float (float64, double)	10	0x0A	1010	8

If, due to the design of a system, variables only require a data type other than those established by the protocol, the new data type can be appended to existing data types or overridden. The data type code must be only 4 bits (0x00 - 0x0F, 0 to 15).

For example, if the system uses a global positioning system, you might be able to append the "Location" data type (or another system-recognizable name) as two 64-bit floats (latitude, longitude); each variable will hold 16 bytes of data, and set the data type code to 0x0B (decimal: 11, binary: 1011). If the entire system is location-based and only requires the "Location" data type, you can set the data type code to 0x00 and omit the other codes.

Then, the protocol doesn't force the user to use the established data types; it only enforces the code size of the data types (codes can only be 4 bits long); the user can append data types or override existing ones.

Connection Types

One-Time Connection

A one-time connection indicates data transfer in request-response mode. This connection is only used when read/write operations are required for long periods of time and/or when a device does not require real-time updates.

When the connection to the broker is established, the connected device must send data within a 60-second interval. If no data is sent during this time period, the server will close the connection.

To indicate this connection, the first byte sent (the most significant byte) must be set to 0xFF (in binary: 1111_1111), and then the request blocks must be sent. Once the broker sends the response, it will close the connection with the device. Requests can only be of two types: GET and UPDATE; and the broker must be configured in normal operating mode or free operating mode.

Request and Response Blocks

A request block consists of a header, an index, and a payload (depending on the request type). The device can send a continuous series of request blocks, and responses will be received in the same order as the request blocks. Response blocks consist of the operation status code and the payload (depending on the request type). Since the blocks are of a predictable size, it is possible to determine the response of the block, making it possible to send the response blocks in the same order in which they were requested. Such that:

Data sent to the server: $\{Code, ReqBlock_0, ReqBlock_1, ..., ReqBlock_N\}$

Response received from the server: $\{RespBlock_0, RespBlock_1, ..., RespBlock_N\}$

Response blocks always maintain the order in which the requests were sent.

Persistent Connection

A persistent connection to the server is useful when a device or client requires real-time updates, since the connection to the server is not lost. To establish a persistent connection, the first byte (the most significant byte) must be set to 0x00. The protocol version, entity type (client or device), keep-alive value, authentication length, authentication key, and device parameters (updates and dependencies) are then declared, following the format:

Byte	**(MSB) Bit 7 – (LSB) Bit 0**
Byte 0	0x00
Byte 1	Protocol Version
Byte 2	- Use Parameter Indexing - Entity Type *See format
Byte 3 – Byte 4	Keep Alive
Byte 5	Auth Length

Byte 6 – Byte (5+L)	Auth Key
Byte (6+L)...	Parameters

* L: Auth Key Length (specified by Auth Length, at byte 5)

*Byte 2 declaration format:

Persistent Conn. Request Byte	**(MSB) Bit 7 … Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 2	-	Use Parameter Indexing	Entity Type		

* bit 3 is boolean: 1 for using parameter indexing, 0 for no parameter indexing.

Entity Type Codes:

Entity	**Code**	**Hex Code**	**Binary Code**
Device	0	0x0	000
Client	1	0x1	001

Keep Alive

Keep Alive is a value that must be set between 60 and 3600. This represents a time interval in seconds during which a connection must send data. If it doesn't send any type of data within the declared time, the server will close the connection. If, according to the design, the device/client doesn't send data (write/read operations, or others), it has the option of sending a single byte called a PING, which the server will respond with a success code (0x00).

Auth Length

Auth Length represents the length of the access key. This access key is defined by the user; the protocol only states that the key has a maximum length of 255 bytes. If this field is declared as 0 (length 0), it will be interpreted as not sending an access key and will proceed to process the

parameters. The characteristics of the access key are defined by the user or broker (depending on the developer); this can be a key, username and password, etc.

Auth Key

The authentication key or access key is a string of characters used to authenticate the connection with the server. The length of this key is declared in Auth Length (previously defined) and may or may not be present in the body of the connection request. The authentication key does not have a defined format; this is defined by the user or the broker.

Parameters

A device's parameters are those that indicate which variables the device updates and which variables it depends on. Each parameter is a block that indicates the parameter type (GET to declare a dependency, UPDATE to declare an update), the index length, and the variable index. They follow the common request header format:

Byte	**(MSB) Bit 7**	**Bit 6 … Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	Method	-	Index Length	
Byte 1 – Byte L	Index			

* L: Index Length (specified by index length, at byte 0)

If the declared device type is client, the parameters are omitted, since clients can write to any variable and receive any update to any variable in real time. Method must be either 0 for a dependency parameter, or 1 for an update parameter.

Parameter Indexing

Parameter Indexing is a data mapping and transmission technique designed to optimize the number of bytes transmitted by a device. Parameters declared by a device are theoretically represented as a matrix with N columns and one row, just like system variables; thus, declared parameters are treated as a set. Each column of the matrix corresponds to a parameter, and each parameter is assigned a unique, continuous index, starting at index 0.

For example, let D_n be a device that depends on the variables (S_d): $\{V_4, V_5, V_6\}$, and updates the variables (S_u): $\{V_2, V_3\}$; definition: $D_n(S_u, S_d); S_u = \{V_2, V_3\}; S_d = \{V_4, V_5, V_6\}$, where $S_u, S_d \in V$. Then each parameter is assigned an index such that:

Parameter	Operation	Variable	Param Index Code
P_0	GET	V_4	0x00
P_1	GET	V_5	0x01
P_2	GET	V_6	0x02
P_3	UPDATE	V_2	0x03
P_4	UPDATE	V_3	0x04

When the connection is successful and parameter indexing is enabled, in addition to the status code, the response will include the index corresponding to each parameter, in the same order in which they were declared, in the following format:

Byte	(MSB) Bit 7 – Bit 2	Bit 1	(LSB) Bit 0
Byte 0	-	Parameter Index Length	
Byte 1 – Byte L	Parameter Index		

* Response format for a indexed parmeter. **L**: Index Length (specified by index length, at byte 0)

The index size is fixed for device operations and depends on the number of parameters declared by the device. The maximum size for a parameter index is 4 bytes (32-bit unsigned integer), and the minimum is 1 byte. This is determined by the number of bytes required to represent the number of declared parameters, in the same way that the length of a variable's index is determined. See *Methods: Length of the Index of a Variable*.

When the device performs an update operation, it should only send the parameter index, followed by the payload. Likewise, when receiving an update to a variable on which the device depends, the broker will deliver it with the parameter index, followed by the payload:

Most Significant Byte	Least Significant Byte
Parameter Index	Payload

Data sending and receiving format when parameter indexing is enabled for a device

Management Codes

Managements codes are single-byte codes sent only by connected devices/clients.

Ping

Keep-alive mechanism. This consists of sending a byte of data to the server to certify that the session remains active. It is particularly useful for maintaining the connection in cases where the device has experienced a long period of inactivity (absence of data traffic) and unilateral termination by the server is required.

Code	0x01
Response	0x00 (Successful operation)

Disconnect

Byte used to request a disconnection from the server.

Code	0x02
Response	NULL, the server disconnects the entity

Methods

Length of the Index of a Variable

The index size consists of 2 bits in the header that indicate the size in bytes of the index payload (which follows the header). 1 is added to the actual binary value to obtain the size in bytes (0 bytes is not allowed), such that:

Binary Value	Length in Bytes
00	1 byte
01	2 bytes
10	3 bytes
11	4 bytes

This data, which indicates the index length, is used to save bytes corresponding to the binary representation of the index, so that the actual value of the index is not lost while its binary representation is compacted.

An index is defined as a 32-bit unsigned integer (range: 0 to $2^{32}-1$). However, to optimize bandwidth usage, the index can be transmitted using only the minimum number of bytes necessary to represent its binary value. The minimum number of bytes required to transmit an index N is determined as follows:

- If N = 0, it is represented by 1 byte (value 0x00).
- For N > 0, the number of bits needed to represent N is:

$$bits(N) = \lfloor \log_2(N) \rfloor + 1$$

- Then, the number of bytes required is:

$$bytes(N) = \left\lceil \frac{bits(N)}{8} \right\rceil$$

In this way:

- $N < 2^8$, (0 to 255) → 1 byte
- $N < 2^{16}$, (256 to 6,535) → 2 bytes
- $N < 2^{24}$, (65,536 to 16,777,215) → 3 bytes
- $N < 2^{32}$, (16,777,216 a 4,294,967,295) → 4 bytes

This ensures that the most compact possible encoding is always used, without losing information.

The user is not required to compress the index size; they can specify that the index is 4 bytes long and represent index 1, such that: (byte 3) 0x00, (byte 2) 0x00, (byte 1) 0x00, (byte 0) 0x01. However, if the index you want to indicate is greater than the declared bytes (e.g., index 456 (2 bytes to represent it) and 1 byte of declared length), then there will be an interpretation problem, since the least significant byte will be taken as part of the variable's value (payload) or as a new request, depending on the used method.

UPDATE STREAM format

The UPDATE STREAM is a format used to indicate the status of a variable. This block consists of the index, the data type, and the variable's value. It is structured as follows:

Byte	**(MSB) Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	-	-	Data Type				Index Length	
Byte 1 – Byte L	Index							
Byte (1+L)	Payload							

* L: Index Length (specified by index length, at byte 0)

Byte 0 is referred to as the header; it must contain the variable's data type and index size. The data type ranges from Boolean to 64-bit decimal and must be specified based on the assigned code. After specifying the data type, the length of a variable's index is declared.

After byte 0 (header), the variable's index follows, the index must have the same length in bytes as declared in the header. After declaring the variable's index, the value of that variable is set. The length of this value matches the data type indicated in the header, as shown in the data type table.

The minimum block length for a V_n variable is 3 bytes, under the following conditions: The variable is of Boolean type (data size of 1 byte), the index is less than 256 (between 0 and 255, 1 byte), plus the header byte: 1 byte (header) + 1 byte index + 1 byte data. The maximum block length for a V_n variable is 13 bytes: 1 byte header, 4 bytes index, and 8 bytes payload.

The UPDATE STREAM is present when an update to the value of a V_n variable is transmitted. This value transmission is sent to all clients and devices that depend on the V_n variable. Like the GET and UPDATE methods, the UPDATE STREAM can be transmitted in a chain (when multiple updates to a variable occur simultaneously).

Each UPDATE STREAM block submission must start with the code 0xFF (most significant byte), similar to that used for read/write operations for one time connections: [0xFF] [UptStr V1] [UptStr V2] ... [UptStr Vn]. The transmission with the start code 0xFF, only apply if the broker transmit a variable update to entities, the start code should not be present when sending a response from a GET request.

Method Codes

Method	Code	Hex	Binary
GET	0	0x00	0000
UPDATE	1	0x01	0001
EXPAND	2	0x02	0010
SET TYPE	3	0x03	0011

- **GET Method**: Method used to get the value of a variable at index N
- **UPDATE Method**: Method used to update the value of a variable at index N
- **EXPAND Method**: Method used to expand the set of variables (Expanding the capacity of the in-memory database)
- **SET TYPE Method**: Method used to set the data type of a variable at index N. Useful when the server is in strict mode; in other modes, the update method can override this value; in strict mode, it is only available for the set type method

Note about expanded mode: Expanded mode indicates that the header is now 2 bytes in size. Expanded mode not only declares the GET and UPDATE methods as normal mode does, but now requires the EXPAND and SET TYPE methods. This expanded mode is only available for client-type entities.

GET Method

To query the value of a variable at index N, the GET method and the length of the variable's index must be specified in the header. The index is indicated after the header. The server must then respond with a status code plus the body of the Update Stream. Bit 7 of the header is set to 0, indicating the GET method. Request structure:

Byte	**(MSB) Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	Method	-	Data Type				Index Length	
Byte 1 – Byte L	Index							

* L: Index Length (specified by index length, at byte 0)

Response Structure:

Byte	**(MSB) Bit 7 – (LSB) Bit 0**
Byte 0	Response Code
Byte 1...	Update Stream Format

Get Method, Expanded Mode

Byte	**(MSB) Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	-	-	-	-	-	Method		
Byte 1	-	-	Data Type				Index Length	
Byte 2 – Byte (1+L)	Index							

* L: Index Length (specified by index length, at byte 1)

UPDATE Method

To update the value of a variable at index N, the UPDATE method, the data type, and the index length must be specified in the header. The index is then declared, followed by the index's new value. The update method has a format similar to Update Stream, with the difference that bit 7 of the header is set to 1, which indicates the update method.

Request structure:

Byte	**(MSB) Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	Method	-	Data Type				Index Length	
Byte 1 – Byte L	Index							
Byte (1+L)	Payload							

* L: Index Length (specified by index length, at byte 0)

The data type is used to calculate the payload length. Depending on the operating mode, if the mode is strict and the data type declared in the header does not match the data type of the variable, then the value will not be updated and an error will be returned. If the server mode is normal or free, then the data type and value of the variable will be updated, and the response will be 0x00 (success).

The response to this request is 1 byte long and indicates the status of the operation, according to the response codes.

Update Method, Expanded Mode

Byte	**(MSB) Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	-	-	-	-	-	Method		
Byte 1	-	-	Data Type				Index Length	
Byte 2 –	Index							

Byte (1+L)	
Byte (2+L)	Payload

* L: Index Length (specified by index length, at byte 1)

EXPAND Method

The EXPAND method indicates the addition of new variables to the set of variables *V.* This method must take a set of data types, which will correspond to the data type of the new variable. The size of this array must correspond to the desired expansion size. If the declared data type is invalid, it will be set to the default data type: INT32. If the server could not expand the database, it will respond with a 0x1E code for each requested variable that follows.

Byte	**(MSB) Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	-	-	-	-	-	Method		
Byte 1 … Byte N	Data Type Array (1 byte per item)							

The set is an array of N bytes, where each byte contains the code corresponding to the variable's data type. This array follows the header (which only indicates the method) and will be responded with a set of response blocks corresponding to the status of each of the data types sent. If the array was of N data types, then the expansion will be of N variables and the response will be of N blocks. There is not any other request after using the EXPAND method, all the remaining bytes are counted as new variable with data type T, part of the array of N bytes.

Response Block:

Byte	**(MSB) Bit 7 – (LSB) Bit 0**

Byte 0	Response Status
Byte 1 – Byte 4	Assigned Index (uint32)

The response is the status of the operation plus the index assigned to the variable. If an error occurred, then the response block will only contain the byte corresponding to the status.

SET TYPE Method

The SET TYPE method indicates the change of a variable's data type. This method, like the expand method, is only available to clients. When changing the data type, the variable's value is set to 0 (the value in its binary representation, all bits will be set to zero) and the value's size is adjusted to the new data type.

Byte	**(MSB) Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**(LSB) Bit 0**
Byte 0	-	-	-	-	-	Method		
Byte 1	-	-	-	-	-	-	Index Length	
Byte 2 – Byte (1+L)	Index							
Byte (2+L)	Variable Data Type							

* L: Index Length (specified by index length, at byte 1)

Response: 1 byte of response per block corresponding to the operation status: 0x00 for successful operation; 0x0D for invalid data type (data type not updated); 0x01 for invalid index; according to the response codes.