# The Integrated Value Communication Protocol

Technical Specification Exposition - Version 1.0

By Franklín Andrésson for **TERRN.**DYNAMICS

IVCP-TSE-0725A

# Table of Contents

# 1. Foundational Principles

## 1.1 What is IVCP?

The Integrated Value Communication Protocol (IVCP) is a single-wire, unidirectional protocol for continuously communicating scaled numeric values between devices. Each message contains an identification-tagged value paired with a checksum validation, allowing devices to target a wide range of specific metrics such as speed, temperature, or torque, in addition to having the receiver verify the integrity of a sent message.

IVCP is designed to facilitate lightweight communications between devices in embedded systems, where simplicity and robustness are critical, rather than the raw rate at which data is handled. IVCP assumes adverse conditions during operation, such as electrically noisy environments or loose connections, and accounts for such variables through its design.

## 1.2 Why was IVCP developed?

IVCP was developed in response to issues personally encountered with traditional methods of transmitting numeric information between devices commonly found in small-scale embedded systems:

- **Pulse Width Modulation (PWM)** is hardware and often software-simple but lacks contextualized capabilities, being unable to single-handedly differentiate between what a transmitted value represents. In addition, it falls short in precision and resistance to electrical noise due to its strictly timing-based nature.
- **Inter-Integrated Circuit ($I^2C$)** is a precision-driven and versatile protocol, but often far exceeds necessary capabilities within the domain of simple numeric transmission, thus making its more complicated setup unfavourable in lightweight embedded systems. Due to its reliance on the clock line, there is also a risk of it freezing and needing manual restarting.
- **Universal Asynchronous Transmitter/Receiver (UART)** is another precise and flexible protocol, but for simple numeric value communication it adds unnecessary overhead and complexity.

## 1.3 Who is IVCP for?

IVCP is intended for a variety of small-scale end-use cases, aimed at being a middle ground between PWM and protocols like $I^2C$ or UART; it can present itself in the hobbyist and university team space, while also proving useful in lighter UAV or motion control projects where precision is desired, but not at the cost of robustness or simplicity.

# 2. Operational Principles

## 2.1 Overview

An analogy can be drawn on the basis of how IVCP operates, namely its likeness to the structured flow seen on train platforms. The transmitter in this case is the train operator, sending trains, messages, at regular intervals down the rail network tracks, in this case, a data line cable.

The receiver in this analogy is a potential passenger on one of these trains. Regardless of the passenger's presence, the trains still run, much like how the transmissions are continuous regardless of whether receivers are present. Once the receiver connects to the data line, like how a passenger enters the platform, it waits for the next message as signalled by a message break. If the receiver misses a message or determines it as corrupt, it returns to its waiting state, scanning for the next break and opportunity to correctly decode the information contained within the proceeding message.

The independence of the transmitter and receiver, where neither relies on the other, and the assumption paired with expectation that messages can be missed and damaged is at the core of IVCPs architecture and operational principles, having shaped its simple yet robust nature.

## 2.2 Message Anatomy

The structure of an IVCP message is critical to its strategic position in efficiently containing and transmitting numeric values. Each message contains 40 bits, transmitted primarily in a return-to-zero (RZ) format, and is designed to maximize simplicity and robustness while conveying a contextualized value.

The following table provides insight into how an IVCP message is structured:

| SEGMENT | BITS | DESCRIPTION |
|---|---|---|
| MESSAGE BREAK **(BRK)** | **2** | 2-bit continuous HIGH to mark the end of one message and the start of the next. |
| VALUE ID **(VID)** | **4** | Tags the given value with an identifier (1-15); 0 is reserved for error handling. |
| SIGN **(VAS)** | **1** | Determines if the value is positive or negative. |
| NUMERIC VALUE **(VAL)** | **19** | Integer mantissa of the value, representing significant digits. |
| EXPONENT SIGN **(EXS)** | **1** | Determines if the exponent is positive or negative. |
| EXPONENT **(EXP)** | **3** | Power-of-ten exponent from 0 to 7, scaling the value. |
| CRC-8 VALIDATION **(VLD)** | **8** | CRC-8 (0x07) checksum for receiver-side message integrity validation. |
| COOLDOWN **(CLD)** | **2** | 2-bit continuous LOW to ensure clean BRK and RZ behaviour. |

*Fig. 2.2A, IVCP Message Segments Table*

This fixed-length message structure is what enables IVCP's contextualized and end-verifiable numeric communication. The omission of explicit start and stop signals is due to their redundancy in the operational style of IVCP. If a receiver connects mid-message, it is aware that it should not attempt to make sense of what is being transmitted until a clear break, a long HIGH, is detected and that it will read a specific amount of bits afterward. Hence, there is no compelling reason to use separate start and stop signals.

The presence of a cooldown is motivated by the design and operational philosophy of IVCP, where robustness is key, and a trailing long LOW will help alleviate any potential electric effects or slight timing issues, allowing for a stable and clean BRK for the receiver to pick up on while reinforcing return-to-zero (RZ) reliability.

## 2.3 Message Transmission

The transmission of messages takes the form of a continuous bitstream consisting of fixed-length messages sent at regular, predetermined intervals without the need for an external clock or feedback from a receiving device, resulting in the omission of procedures such as pausing, retrying, or re-sending data. Every individual transmitted message starts with a 2-bit continuous HIGH that is exempt from return-to-zero (RZ) logic, forming a unique signature to which a receiver can latch since it cannot be confused with the regular 1-bit, RZ-logic regulated HIGHs.

Each bit spans a constant time period referred to as T:

- **A binary "1"** is transmitted as HIGH for 0.5T followed by a LOW of 0.5T.
- **A binary "0"** is transmitted as a continuous LOW of 1.0T.

The return-to-zero (RZ) rule applies to every bit following the BRK segment, with its timeline being offset 0.5T. The different timelines can be summarized as follows:

- **The bit-aligned timeline** is established 1.0T after BRK drops LOW. A sustained LOW is held at least 0.75T after the initial drop of BRK to satisfy RZ logic, after which the signal either goes HIGH (binary "1") or LOW (binary "0") in time for the first bit.
- **The RZ-aligned timeline** is established 0.5T after BRK drops LOW, in matching but offset 1.0T intervals from the bit-aligned timeline.

Once 38 bits, each spanning a full period of T, have been transmitted as required, a 2-bit continuous LOW period is held to preserve electrical stability and ensure signal integrity, providing a clean runway for the next BRK. With those 2.0T of LOW, the 40-bit message concludes and yields clean separation before the next transmission occurs.

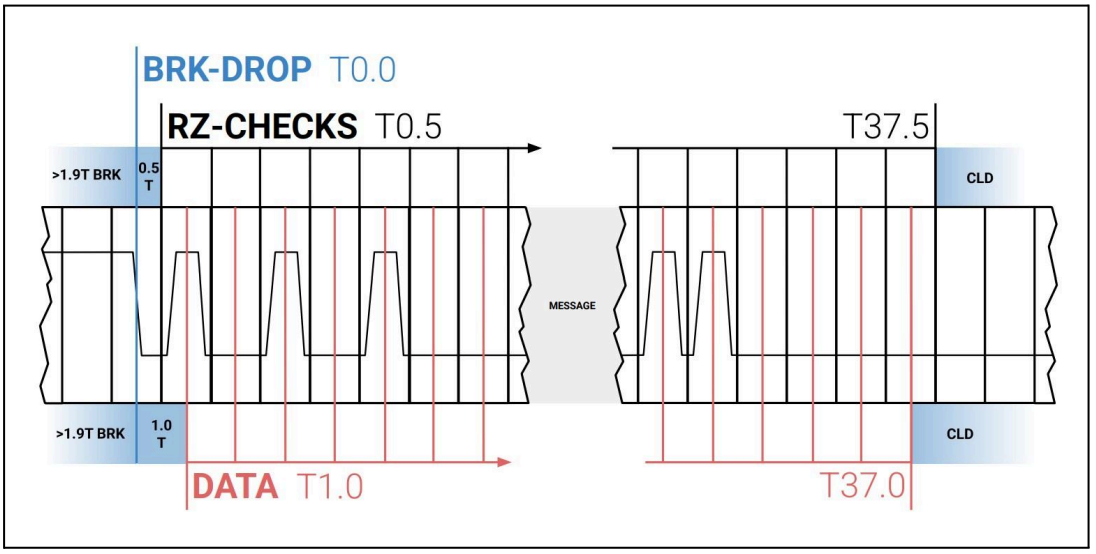


*Fig. 2.3A, IVCP Transmission Timelines*

## 2.4 Message Reception

The reception of messages is done in a self-syncing manner, where the receiver waits to latch onto a BRK, knowing a fixed-length message will follow. Once BRK goes low, it is aware that there are 36 payload bits to process, excluding the 2-bit BRK and CLD framing the message. Once the information has been read, the receiver goes back to listening for a BRK, making this protocol flexible for slight timing errors and clock misalignments that would otherwise accumulate over extended periods of operation.

The receiver is responsible for managing two timelines, namely, the aforementioned bit-aligned and RZ-aligned timelines. Every 0.5T it samples the signal state (HIGH or LOW), alternating between data bit reads and RZ checks. The following procedure is performed by the receiver:

- **The receiver listens** and measures every HIGH pulse, where any pulse longer than 1.9T is interpreted as a BRK.
- **Once a BRK** has been detected, the receiver waits until BRK drops LOW.
- **0.5T after LOW,** an RZ-check is performed and its subsequent timeline is established.
- **1.0T after LOW,** the first data bit read is performed and its subsequent timeline is established.
- **37.0T after LOW,** the last data bit read is performed, finalizing the message.
- **37.5T after LOW,** the last RZ-check read is performed, preserving electrical integrity.

The decoding process involves splitting the 36-bit message into its corresponding segments as described in the segment table in section 2.3. The 8-bit segment at the end of the data bits is the VLD segment, where the receiver can validate the integrity of the preceding 28 bits using an 8-bit Cyclic Redundancy Check (CRC-8). This checksum is calculated by the transmitter and verified by the receiver against the interpreted message. If any of the following conditions are met, the receiver discards the message and returns to its listening state:

- **An RZ-check violation,** where a signal state hasn't returned to zero, results in an immediate return to listening.
- **A failed CRC-8** results in the receiver ignoring and discarding the message.
- **A VID of 0** results in an immediate return to listening, addressing issues found in early testing of IVCP where empty messages were accompanied by VIDs of 0.

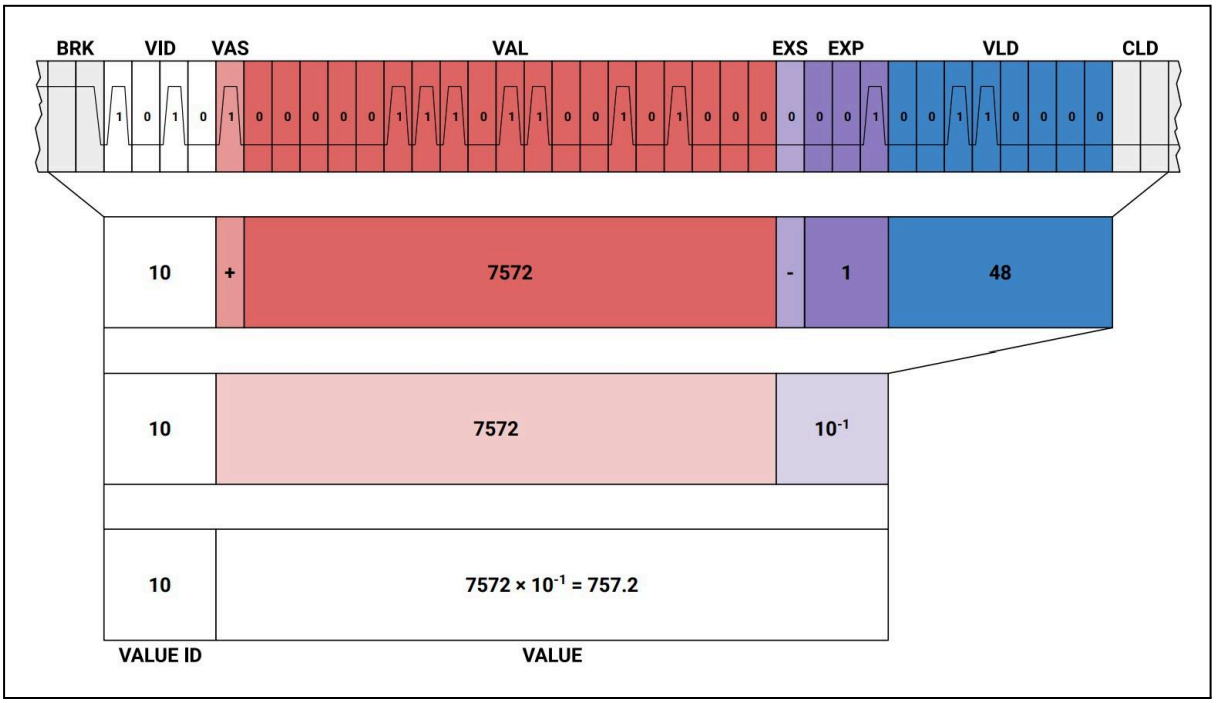


*Fig. 2.4A, IVCP Receiver Message Decoding & Data Extraction*

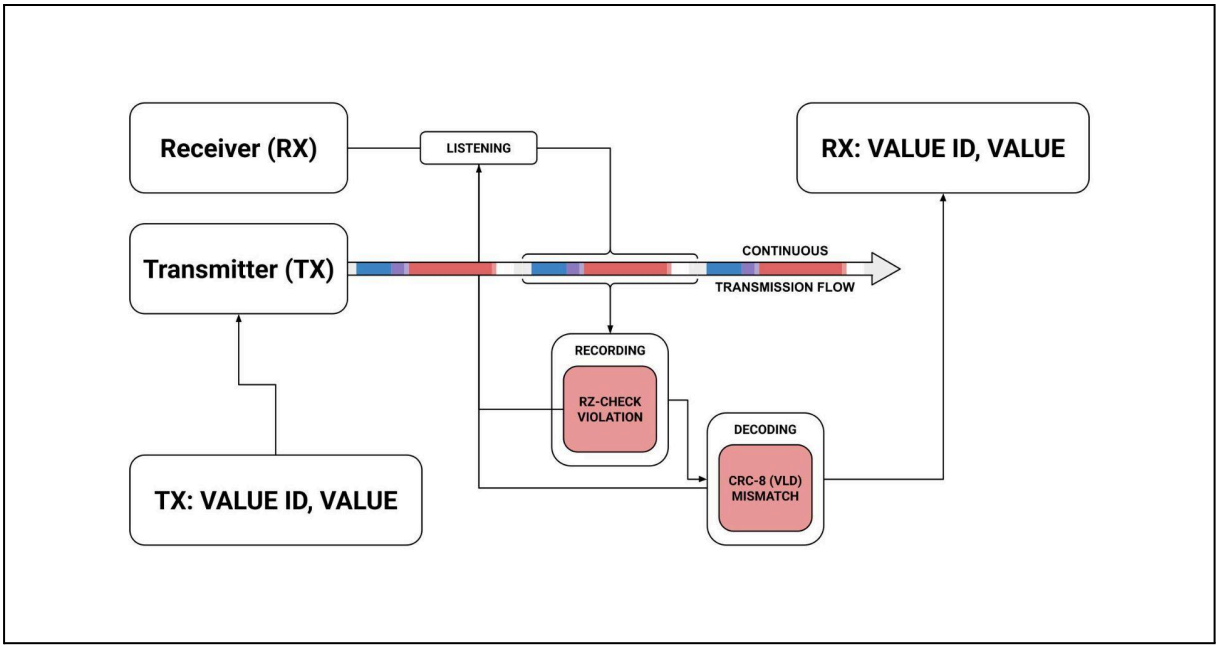


*Fig. 2.4B, IVCP Transmitter-Receiver Relationship & Message Handling*

## 2.5 Integrity by Design

The design of IVCP is meant to provide generous margins and graceful error handling, valuable in sub-par operational environments. The protocol operates without an external clock, calibration, or feedback from the receiver, allowing the transmitter and receiver to operate independently, without making either rely on the other.

Through the use of fixed-length messaging and bit timing, paired with the receiver returning to its listening state after every message concludes, the protocol re-syncs at the same rate as messages are sent, eliminating issues related to clocks drifting over time and allowing for disruptions. The unique BRK signal is purposefully designed to never be confused with a data-HIGH.

The return-to-zero (RZ) logic reinforces integrity. Inspired by aviation protocols to the likes of ARINC, it allows for the monitoring and enforcement of electrical signal integrity, detecting if the data line is being held HIGH and the subsequent cancellation of message reading. Paired with RZ logic is the CRC-8 validation that serves as a data integrity verification. Just as the RZ-check, a faulty CRC-8 results in the receiver's return to the listening state.

For the multiple layers of error detection, there is also a graceful error handling principle. In the environments that IVCP is designed to operate and prove useful in, especially motion control-heavy cases such as UAVs, the idea of "the show must go on" is very important to the continued operation of equipment even in sub-par conditions. In certain settings it is far from practical that a protocol can freeze up or stagnate in cases such as failed handshakes or corrupted messages. Therefore, the design of IVCP already assumes these things will happen and reflects it in how it simply ignores and discards faulty messages and waits for the next one.

This is possible due to IVCP operating a continuous flow of data, so it doesn't really matter if a message is missed from time to time since there is always a new one right after. This is demonstrated in its hot-plugging capabilities, being able to let go of missed messages in a sudden disconnect and picking right back up once it is connected again without any manual start or restart.

# 3. Specification

## 3.1 Electrical Requirements & Recommendations

For the protocol to work, one data cable and one ground connection is needed. The voltage threshold many vary on a per-device basis, but for the Raspberry Pi Picos that were tested as transmitter and receiver the following applied:

- **Input LOW** ≤ 0.8V
- **Input HIGH** ≥ 2.0V

Due to the testing device, the logic level is 3.3V. This can however work on any practical voltage due to IVCP being based on absolute HIGHs and LOWs that devices classify individually. The arrangement of which IVCP can be connected physically is up to the operator. Examples are shown below:

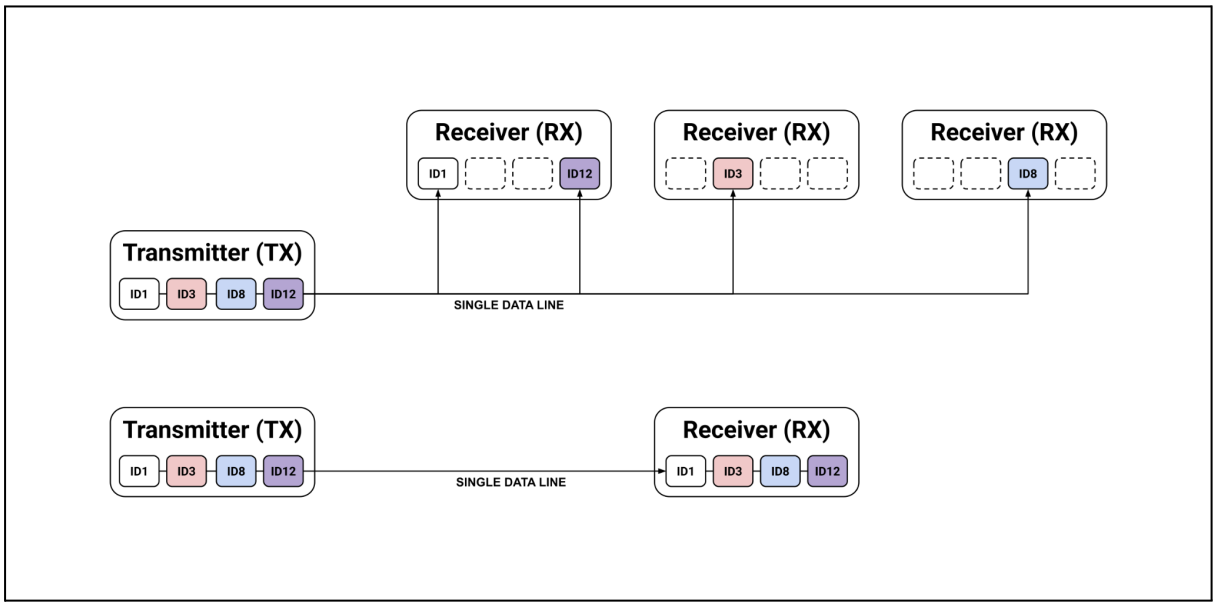


*Fig. 3.1A, IVCP Transmitter-Receiver Setup Example*

As can be seen in the diagram, IVCP is designed with multifunctionality in mind, fulfilling demands such as but not limited to:

- **Single-TX, Multi-RX** where receiving devices share the same data line but only register values that correspond to specific value ID's. Receivers can also be set up to accept more than just one ID as shown.
- **Single-TX/RX** that carries an array of different value ID's. Useful in communicating values that correspond to different metrics such as speed, torque, and temperature.

## 3.2 Utilization & Integration Examples

IVCP has been designed to be as easy to integrate and use as possible. This is achieved through the microPython `ivcp.py` library from where you can get two classes, namely `Tx` and `Rx`. Within them, you find the following which you can use:

- **`Tx.transmit(value, vid)`** is the common method of the tx object. It takes two parameters, the value that needs transmitting (value) and its corresponding id (vid), and continuously transmits them.
- **`Tx.stop()`** is a method of the `tx` object. It stops the transmission brought by `tx.transmit()`.
- **`Tx.blip(value, vid)`** is a method of the `tx` object. It takes two parameters, the value that needs transmitting (`value`) and its corresponding id (`vid`), and transmits them once.
- **`Rx.listen()`** is a method of the `rx` object. It starts a background thread that runs continuously, decoding every incoming message.
- **`Rx.grab()`** is a non-blocking method of the `rx` object. It returns the latest valid (`value`) and its corresponding id (`vid`), able to be used in, for example, calculations. It is important to note that the background thread has to be active. If no message is decoded by the time `rx.grab()` returns `(0.0, 0)`, preventing crashes in programs that might use the value in calculations.
- **`Rx.status()`** is a non-blocking method of the `rx` object. The following diagnostic codes are returned by this method when called upon:
    - 1: Last message reception attempt was OK.
    - 0: VID = 0, last message reception was discarded.
    - 2: CRC-8 mismatch, last message reception was discarded.
    - 3: RZ violation, last message reception attempt was discarded.
    - 4: No message received yet.
- **`Rx.stop()`** is a method of the `rx` object. It stops the background thread that runs continuously.

To use these instances, one must first import `Tx` and/or `Rx` from `ivcp` and define the transmitting or receiving pin through the following:

- **`tx = Tx(data_pin, t_value)`** where data_pin is the gpio pin and t_value is T.

Below are examples of IVCP program implementation.

**Random Value & Value ID Transmission Loop:**

```
from ivcp import Tx
import utime
import random

tx = Tx(15, 8000) # (data_pin, t_value in µs)

while True:
    rval = random.randint(-1000, 1000)
    rvid = random.randint(1, 15)
    tx.transmit(rval, rvid)
    utime.sleep(0.1)
```

**Received Value & Value ID Adder Loop:**

```
from ivcp import Rx
import utime

rx = Rx(15, 8000) # (data_pin, t_value in µs)

rx.listen()

utime.sleep(0.1) # Time to allow for message reception

while True:
    value1, vid1 = rx.grab()
    utime.sleep(0.5)
    value2, vid2 = rx.grab()
    print(value1, "+", value2, "=", value1+value2)
```

**Basic Time-Restricted Transmission**

```
from ivcp import Tx
import utime

tx = Tx(15, 8000) # (data_pin, t_value in µs)

tx.transmit(123.45, 10) # (value, vid)

utime.sleep(1)

tx.stop()
```

**Basic Continuous Transmission**

```
from ivcp import Tx

tx = Tx(15, 8000) # (data_pin, t_value in µs)

tx.transmit(123.45, 10) # (value, vid)
```

**Basic One-Shot Transmission**

```
from ivcp import Tx

tx = Tx(15, 8000) # (data_pin, t_value in µs)

tx.blip(123.45, 10) # (value, vid)
```

**Basic One-Shot Reception**

```
from ivcp import Rx
import utime

rx = Rx(15, 8000) # (data_pin, t_value in µs)

rx.listen()

utime.sleep(0.1) # Time to allow for message reception
value, vid = rx.grab()
print(value, vid)

rx.stop()
```

**Basic One-Shot Reception Status**

```
from ivcp import Rx
import utime

rx = Rx(15, 8000) # (data_pin, t_value in µs)

rx.listen()

utime.sleep(0.1) # Time to allow for message reception
rxstatus = rx.status()
print(rxstatus)
# STATUS: 1=OK, 0=VID_FAULT, 2=CRC_FAIL, 3=RZ_FAIL, 4=NO_MSG

rx.stop()
```

## 3.3 Performance & Benchmarks

The following performance figures were achieved using two Raspberry Pi Picos connected over jumper cable between their respective GPIO pins number 15.

The testing was done through having an RX demo program listen for a transmission from a connected TX and start decoding it immediately. Both programs used `ivcp.py` as the library for the functioning of the protocol.

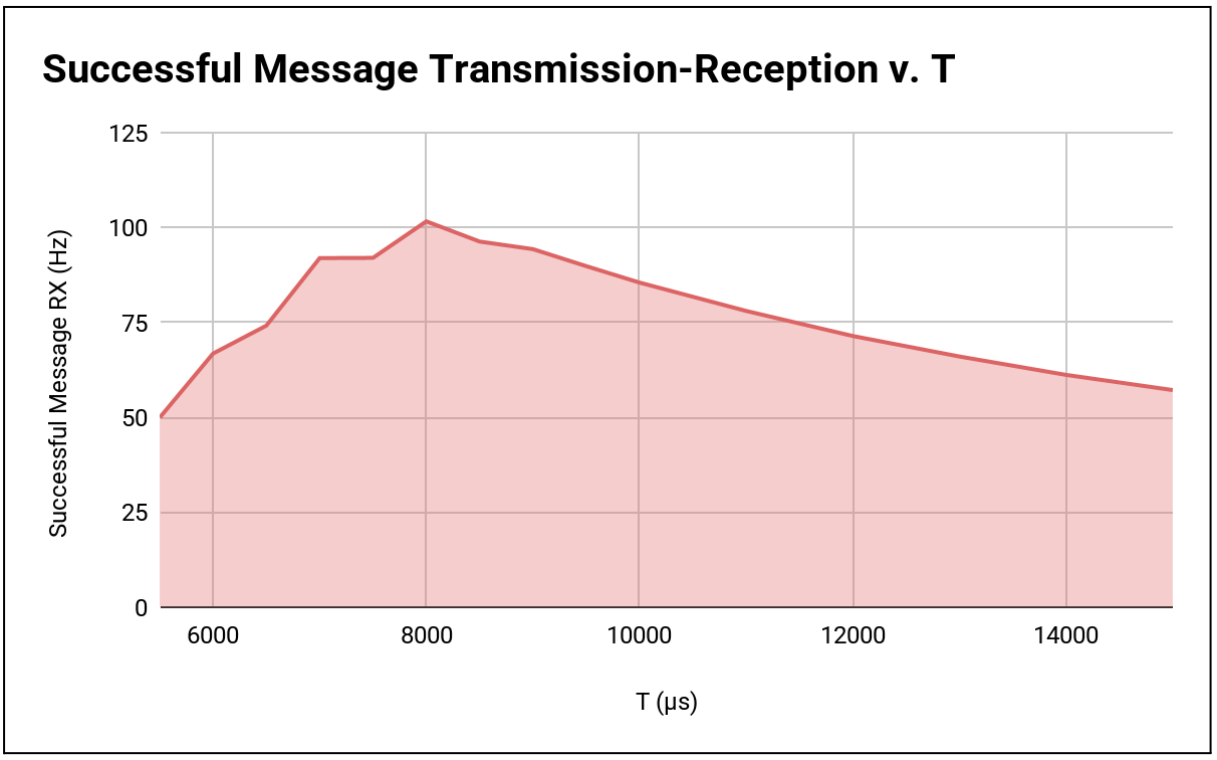


*Fig. 3.3A, IVCP RPi Pico Message Rate Performance Graph*

As can be seen in the graph above, the recommended T for running IVCP between two RPi Picos is 8000 µs, peaking at an average successful message rate of 101.6 Hz, where you get the most information per time unit for the least amount of errors.

Beyond a T of 8000, errors started presenting themselves, with almost all of them regarding a failed VLD (CRC-8) segment. It is important to re-state that errors are always discarded by the receiver so even though a T of 12000 comes with almost no errors whatsoever, the amount of information that is correctly handled still peaks at 8000, with any less T resulting in an increasing amount of errors that are then not counted as correctly handled.

It is critical to note that during no part of the testing that involved thousands of messages across different Ts, no wrong values were received as correct, instead being discarded.

# 4. Limitations

Due to the simplicity of this protocol, there are features intentionally left out to preserve IVCP's core principles. These design-rooted restraints and current technical limitations are listed below.

- **Unidirectionality** is an integral part to IVCP's architecture, the key to the relationship between the transmitter and receiver. This does present challenges in two-party validation however, where the transmitting device cannot confirm that the receiver read the message correctly or is even active.
- **No collision or arbitration** makes multiple transmitters on a single data line produce errors for the receivers, this is again to preserve the simple and robust nature of IVCP.
- **Fixed-length messages** make the receiver only receptive to 19-bit mantissas, meaning there is no dynamic sizing of messages, a result of opting for simpler protocol structure.
- **Addressing** is limited to 15 individual ID's, due to 0 being excluded for error detection.
- **No dropped-message** acknowledgment from the receiver to the transmitter due to IVCP's unidirectionality.

# 5. Future Considerations

The current specification of IVCP is early and a proving ground for future versions. The following are features and refinements in mind that would either come in the form of a version or category of IVCP:

- **Time since the last valid message** is an idea for an instance of the `rx` class, where the user could call for how much time has passed since the last valid message reception, enabling programs to identify and act on faulty connections.
- **Decreasing T** is desired to make IVCP more favourable in precision motion control applications. This would have to be achieved through changing the way hardware and software is used by the library, and maybe move to a different programming language entirely from microPython to remove overhead.
- **Enhanced error detection or correction** through known techniques, very useful to be able to correct for messages instead of dropping them entirely.