Experience and Investigation of Data Acquisition Optimisation and Data Storage Methods

Nathan Truong
Level 3 MSci. Laboratory, School of Physics, University of Bristol.
March 25, 2025

***Abstract*—Many scientific experiments rely on data acquisition (DAQ) systems to collect and process measurement data. These systems enable the recording, analysis, and interpretation of various physical or environmental parameters, regardless of the scale of the experiment. This technical report evaluates various methods of data communication using the Raspberry Pi Pico, an affordable microcontroller. The methods evaluated consisted of sending the data through a human-readable format or through binary. Various avenues of optimising the latter were also explored. In addition, different methods of storing the data received by the Raspberry Pi Pico were examined. The primary storage methods examined were CSV, JSON, BIN, and HDF5. Two variations of BIN files were considered: structured BIN, implemented using Python's `pickle` module, and unstructured BIN. Each method was examined for their read and write speeds, as well as how file sizes scaled with the shape of the stored data.**

I. Introduction

A. Data Acquisition Systems

In many scientific experiments, a real-world physical quantity has to be measured to assist in drawing conclusions. Many physical phenomena are regarded as analogue signals. Common examples being temperature, brightness, pH and more. Data acquisition (DAQ) systems would all have at least a sensor and an analogue-to-digital converter (ADC). The sensor is used to capture the characteristics of the system (analogue signal) at a given point in time. However in order to quantify the characteristic, the ADC must convert the analogue signal into a digital signal for a computer to read and translate into a value. DAQ systems can have various characteristics. Some characteristics involve, but are not limited to, capability of managing large amounts of data, high-speed connection to the DAQ, digital recording, and the ability to be fully reconfigured [1].

This report focuses on optimising data transmission from a Raspberry Pi Pico to a DAQ computer, exploring various data formatting and storage techniques, and evaluating methods for reading and storing data in various file formats. Several experiments involving a DAQ system require the results of these investigations to be considered. One example being the CMS experiment at the LHC for data-taking run-2. In addition to obtaining new hardware to upgrade the DAQ system, an investigation of alternative methods to write data to files at high speeds was also performed [2].

B. Analogue-to-Digital Converters

ADCs convert analogue signals into digital signals, enabling computers to process the data. This requires discretising the measured physical quantity and representing it in binary format. The simplest and fastest ADC is the Flash ADC. The general circuitry of a Flash ADC is comprised of a series of resistors which are each connected to comparators. For two resistors, each resistor is connected between positive and negative terminals on the comparator. If the input voltage of positive terminal is larger than that of the negative terminal, this would produce a digital signal of 1. For the opposite case, this would produce 0. The final output is produced by an encoder which configures the combination of 0's and 1's and would provide the digital signal outputted in binary [1].

II. Methodology and Tools

A. Raspberry Pi Pico

The device chosen to obtain physical readings was the Raspberry Pi Pico. The Raspberry Pi Pico is a microcontroller with a motherboard that contains a plethora of features such as a temperature sensor, real-time clock (RTC), four 12-bit ADC outputs

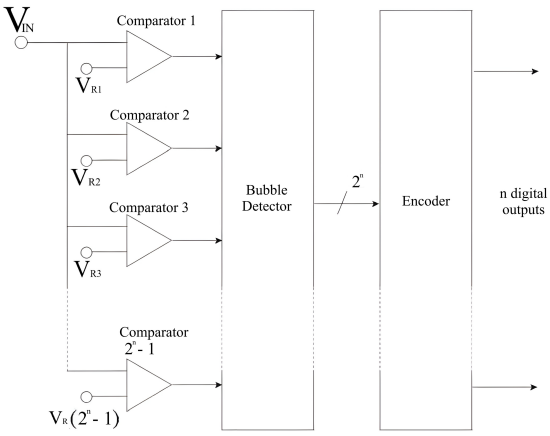

Fig. 1. Circuit diagram of a typical Flash ADC. Taken from [1]

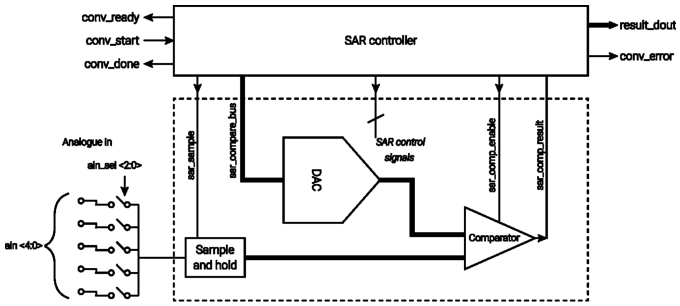

Fig. 2. The circuit diagram of the SAR ADC in the Raspberry Pi Pico. Taken from [3]

(three if excluding the temperature sensor), dual-core ARM Cortex M0+ processor (”RP2040”) and more [4]. The Raspberry Pi Pico was the choice of microcontroller to conduct the investigation due to it featuring an internal clock, which enables time tracking, and its sufficient number of ADC channels, which were enough for the scope of the investigation. Additionally, its cost effectiveness makes it an ideal choice to facilitate the reproducibility of results and allow more peers to conduct this investigation if they desire [5].

The internal ADC within Pi Pico is a Successive Approximation Register Analogue to Digital Converter (SAR ADC). Its construction is similar to the Flash ADC from figure 1. However, the key difference lies in the use of the successive approximation algorithm in the SAR ADC, where the input voltage is compared bit by bit to produce a sequential digital approximation, rather than generating a direct single-step output as seen in Flash ADCs [6]. Most personal computers keep track of time by counting seconds since January 1st, 1970 (the epoch), even when powered off, using a battery-backed real-time clock. In contrast, the Raspberry Pi Pico does not have a battery. Instead, it sets the epoch to 0 and starts counting the moment it is powered on. The Raspberry Pi Pico can be programmed using C/C++ to transfer data to a data acquisition device.

B. Data Acquisition Device

To receive data from the Raspberry Pi Pico, the Raspberry Pi Pico was connected to an Acer Swift X - SFX14-41G laptop (DAQ computer) installed with Windows 11. It received data from the Raspberry Pi Pico through a serial via USB connection. To capture the data received by the Raspberry Pi Pico, multiple Python programs were developed, each tailored to store data based on the type of data received and the corresponding file format. This approach offers the added benefit of minimising performance bottlenecks that would otherwise arise from using Python if statements to switch between different experimental scenarios.

C. Experimental Methodology

One of the main experimental objectives was to investigate into methods which the Raspberry Pi Pico can send data to the DAQ computer at the fastest rate. To find the rate which data is sent, the average difference between consecutive timestamps was calculated as the metric of the rate. The data sent by the Raspberry Pi Pico consisted of several combinations: a timestamp with temperature, a timestamp with temperature and CPU clock speed, and a timestamp with temperature, CPU clock speed, and voltage readings from ADC0. These combinations represent the number of columns of the data, in which the number of columns would be 2, 3 and 4 respectively. To convert from an ADC voltage reading V_{ADC} to a temperature value T in Celsius, the RP2040 data-sheet states the formula for calculating this [3],

$$T = 27 - \frac{(V_{\text{ADC}} - 0.706)}{0.001721} \,. \tag{1}$$

The first initial method used to send the Raspberry Pi Pico readings to the DAQ computer was through sending a string of characters containing the readings. Captions were added for clarity; generally, in the format of `Onboard Temperature`

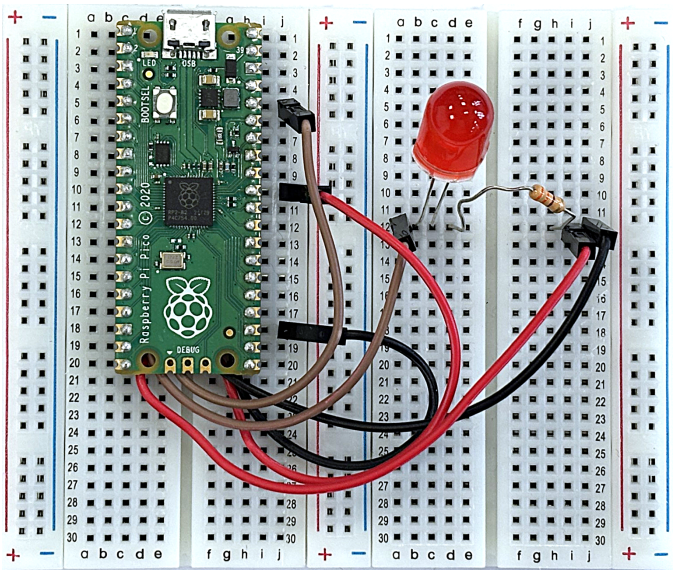

Fig. 3. Circuitry used to obtain the voltage readings from ADC0.

`@ [timestamp] = [temperature] C`. The second method removes the clarity of what is being sent by the Raspberry Pi Pico by simply outputting only the numbers separated by a comma. The CPU clock speed and ADC readings were concatenated at the end of the string, separated by a comma, with this method. The last method removes all forms of clarity by sending the data only through raw binary.

One of the other main experimental objectives was to investigate the various data storage methods through various file types. The investigated file types were comma separated values (CSV), JavaScript object notation (JSON), binary (BIN) and Hierarchical Data Format Ver. 5 (HDF5). Two types of binary files were also investigated: structured binary files constructed with a Python module known as `pickle` and sequentially reading a fixed number of bytes from unstructured binary files that can be converted into the corresponding decimal values. For example, if an `unsigned long long integer` and a `float` was being sent by the Raspberry Pi Pico, the DAQ computer would sequentially read the first 8 bytes and convert them into a decimal value, followed by reading the next 4 bytes and performing the same process. This process would continue, repeating for each subsequent set of bytes, until the end of the file is reached. The latter was carried out with the assistance of the `struct` module. Arbitrary test data was created for this part of the investigation. Data stored from writing the messages from the Raspberry Pi Pico could not be used as JSON files can not support `unsigned long long integers`. The sequence of data types chosen for this investigation was `unsigned integer` and `double precision float`. This pattern alternated according to the column size tested. The main quantities investigated were the read and write times for each file, along with the file sizes, and how these variables scaled with the structure of the data. To fulfil this experimental objective, the Python programming language was used because it is open source with a plethora of modules that can be used to interpret the listed files. It was also a popular programming language at the time of writing for performing data analysis.

III. Benchmark Results and Discussion

A. Data Transfer Rates

The average difference in timestamp obtained when sending the human-readable string with timestamp and temperature was found to be 3720 ± 0.040 , which translates to $260 \pm 3.00\ Hz$. Figure 4 shows the distribution of the difference in timestamp and shows that there was little deviation between readings.

Upon removing the characters clearly indicating what is sent, the average difference in timestamp saw a decrease in 53.0% . The average difference in timestamp from the Raspberry Pi Pico writing its clock speed and voltage from ADC0 is also less than that of writing only timestamp and temperature including the `Onboard Temperature etc.` string. The decrease in difference would be due to the reduced number of characters sent from the Raspberry Pi Pico through omitting the mentioned string. The distribution of differences in timestamp is shown in figure 5.

The difference in timestamp, scaling with the growth in digits, was also investigated. It was found that as a new digit was added to the timestamp, the difference in timestamp would increase by $6.36 \pm 0.30\%$ on average. This was calculated by finding the average difference in timestamp by the number of digits in the timestamp $(d + 1)$ and taking the ratio between the average difference in timestamp (d). With this ratio denoted as (S_d),

$$S_d = \frac{\text{Mean Time Difference at } d+1}{\text{Mean Time Difference at } d} . \quad (2)$$

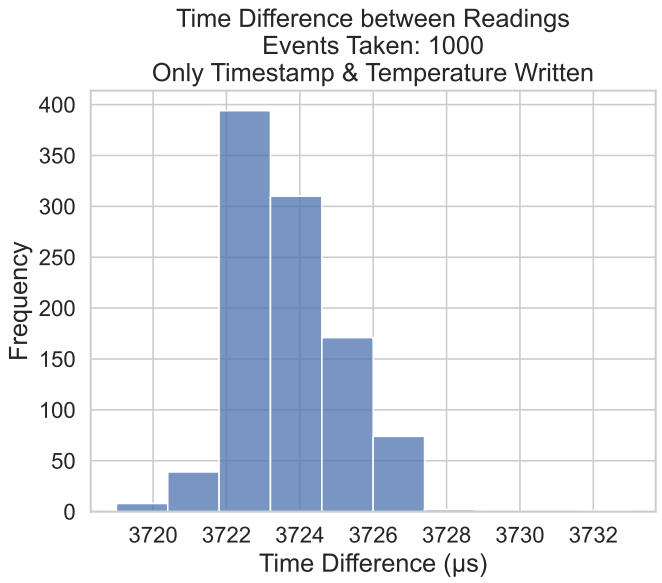

Fig. 4. Histogram of the difference between consecutive timestamp readings from the Raspberry Pi Pico sending the human-readable string.

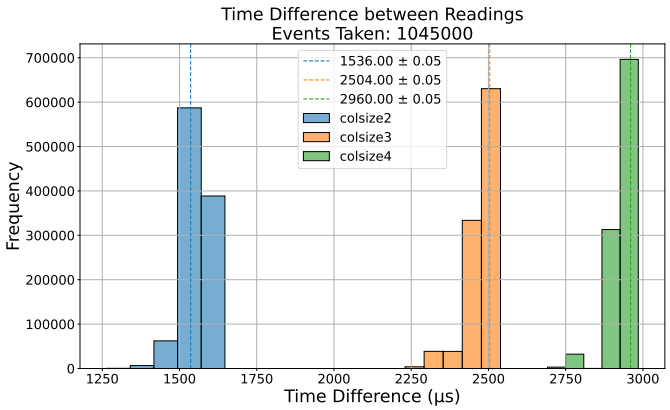

Fig. 5. Difference in timestamp readings from sending only the figures separated by comma. Data was saved to multiple CSV files. The average timestamp difference is represented by a vertical line on the plot, with the exact value displayed in the legend.

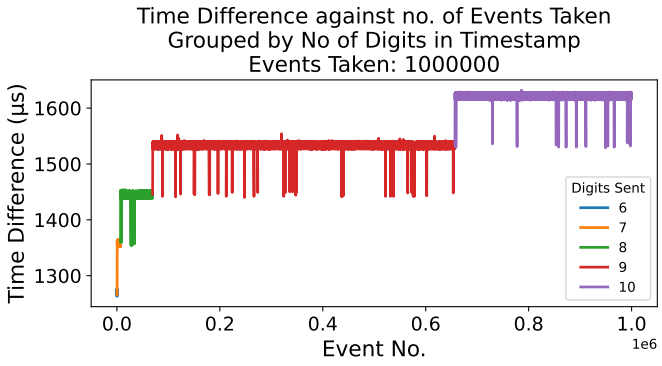

Fig. 6. Difference in timestamp grouped by number of digits in the timestamp. The increase in time difference from an event before the timestamp increases in digits is instantaneous in the next event.

The Raspberry Pi Pico sending data in raw binary provided the smallest difference in timestamp readings/largest frequency that data was transmitted. Compared to sending natural digits, sending timestamp and temperature with raw binary saw an average decrease of $\approx 99\%$ in the average difference in timestamp.

Initial attempts to capture raw binary readings were unsuccessful, as the captured bits from the DAQ computer would not align with the start of the data transmission from the Raspberry Pi Pico. This was noticed from obtaining comically nonsensical timestamp and temperature readings from reading the output binary files. To ensure synchronisation between the Raspberry Pi Pico and the DAQ computer, a trigger system was implemented. A predefined `start-of-message` bit pattern was transmitted, allowing the DAQ system to detect the beginning of the Raspberry Pi Pico's message before writing the received binary to the BIN files. The chosen trigger was a single-byte sequence, `0xAA = 10101010`, which provided a distinct and easily recognisable pattern for data alignment while minimising writing overhead to maintain a high data transmission rate.

Three different timestamp difference readings were obtained from the Raspberry Pi Pico using different data transmission methods. The first involved sending the timestamp along with the temperature, the second involved transmitting the timestamp with the raw ADC reading, omitting the Raspberry Pi Pico from using equation 1, and the third method buffered the timestamp and raw ADC readings in the Raspberry Pi Pico's RAM before transmitting all the data in bulk.

Figure 7 illustrates the distribution of differences in timestamp from the Raspberry Pi Pico sending raw binary. From the figure, it can be observed that the timestamp and raw ADC readings buffered to memory before being transmitted to the DAQ computer has no beneficial effect on the frequency that data is sent. The Raspberry Pi Pico omitting equation 1 in general observes a $\approx 25\%$ decrease in the average difference in timestamp.

With a comparison of how the frequency of data transmission evolved with the number of recorded events, alongside the impact of the timestamp digit length, the difference in timestamp did not deviate from the average beyond $\approx 17\%$ for all scenarios when raw binary was sent. Putting all the scenarios

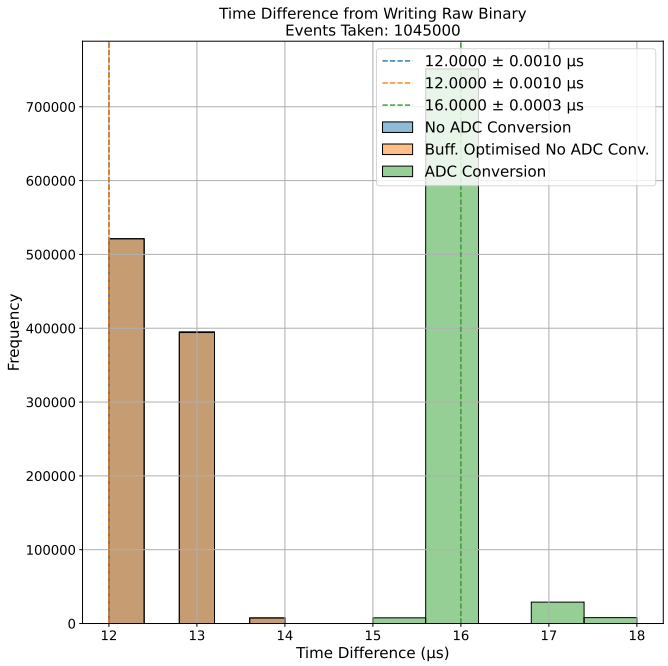

Fig. 7. Histogram comparing the difference in timestamps between sending the raw ADC value directly from the Raspberry Pi Pico and sending the ADC value after conversion. 1045000 events was recorded in total. The average timestamp difference is represented by a vertical line on the plot, with the exact value displayed in the legend.

into perspective, the average difference in timestamp when the Raspberry Pi Pico sends raw binary is ≈ 2 orders of magnitude lower than when sending the actual digits. Therefore, the frequency deviation is smaller when sending raw binary compared to sending character digits.

Another reason for the significant increase in data transmission frequency when sending raw binary is that transmitting character digits requires sending more bits. Fundamentally, an arbitrary character, including a digit, requires multiple bits to translate itself into computer-readable bits. This translation layer is known as a character set. The method used to transmit character digits to the DAQ computer involves encoding each character using the UTF-32 character set [7]. This would mean that for each character, the Raspberry Pi Pico would need to send 32 bits to represent one character. However, with sending raw binary, the number of bits does not fluctuate even as the number of digits that make up the timestamp grows. This is because the timestamp and temperature are represented using data types with a fixed number of bits. Specifically, the timestamp was represented by the `uint64_t` type which uses 64 bits. The temperature was represented by the `float` type if equation 1 was applied and by the `uint8_t` if the equation was omitted. This would mean that even sending the string of characters of length 3, for example to represent "247", 96 bits is required to send this string. Whereas if 247 was interpreted as a `uint64_t` type, only 64 bits are required to be sent as 247 would simply be represented as `11110111` (omitting the leading 0's for readability).

Despite the $\approx 99\%$ decrease in average difference in timestamp, some rows were found to be not interpreted correctly and output non-sensical timestamps or ADC reading values upon being translated from binary despite implementing the trigger system. Non-sensical timestamps were identified if within the next timestamp, the magnitude has increased by any value beyond the 75th quartile. Non-sensical ADC readings were identified by applying equation 1 after data collection and observing any illogical readings. Additionally, for both quantities, negative readings were observed and interpreted as non-sensical. Figure 8 illustrates how the number of non-sensical rows scales with the number of events taken in a given series of measurements. The average percentage of non-sensical rows to the amount of rows recorded was found to be $24.4 \pm 2.00\%$ with the conversion of the ADC voltage, $10.3 \pm 2.00\%$ without converting the ADC voltage and the same result was obtained from buffering all the readings to memory before transmission.

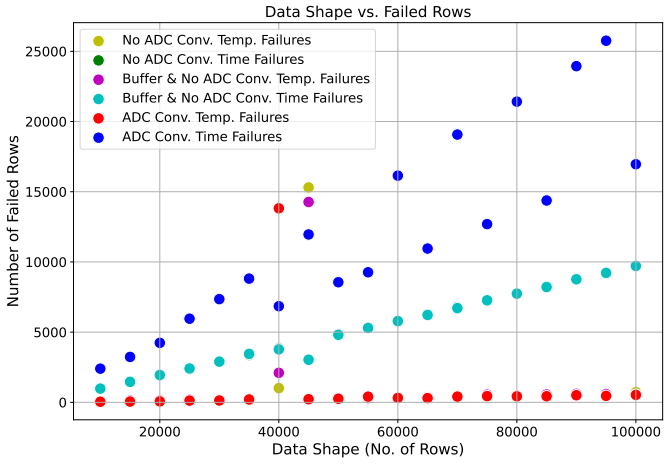

Fig. 8. Number of non-sensical rows in comparison to total number of rows recorded (data shape). The number of failed rows in both scenarios where the ADC reading did not have equation 1 applied by the Raspberry Pi Pico deviate by more than 2%. Hence the lack of green points in the figure.

B. File Storage Methods

Figure 9 illustrates the time taken to convert data from various file formats to a human-readable format. Initially, raw binary was expected to be the fastest format for reading data, as it avoids the overhead of character encoding. However, reading binary still requires parsing a fixed number of bits and converting them based on their intended data types, introducing additional processing time.

The file format with the longest read time was found to be the JSON file. JSON files are comprised of key:value pairs in between curled braces {} separated by a comma. The Python `json` module has functions that can read the contents of JSON files and deserialise the contents into Python objects [8]. This process is slow due to the strict complex structure that JSON files must follow. The JSON file specification uses six structural characters (‘[,’ ‘,’ ‘],’ ‘,’ ‘:,’ and ‘,’), defining and delimit the locations and structure of its contents [9]. In comparison, the CSV file only uses two, ”,” and ”\n”.

Reading a binary file created through `pickle` observed the second fastest read times. `pickle` observes similarities to the Python `json` module, with the main difference being that it uses more ”compact binary representation” rather than text [10]. This would mean that fewer bits would need to be interpreted to convert the contents into human-readable decimal values.

HDF5 files observed the fastest read times. While it also stores the contents of its data in binary, it uses ”a highly efficient metadata map” as its data structure [11]. This metadata map could possibly be attributed as to why it observed the fastest read times.

Figure 10 illustrates the time taken to write to various data types. It can be observed that the time it takes to write to a JSON file is a magnitude larger than all the other file types. This is due to reasons similar to reading larger JSON files. With larger files, many more structural characters are needed compared to the other file types. From figure 11, the CSV file is the file that exhibits the highest write times on average. It is also the only text-based file format in figure 11. This would mean that more bits on average would be needed to represent a record compared to binary and HDF5.

Binary files written with and without `pickle` show similar write times on average. The time

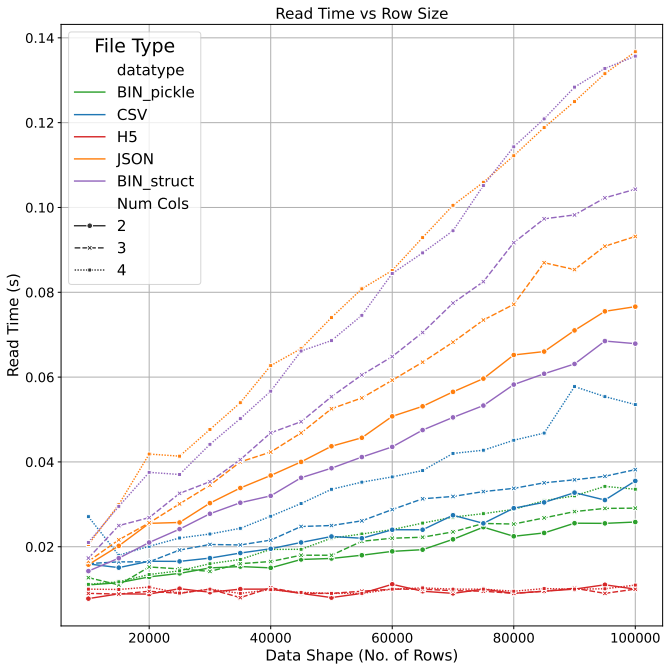

Fig. 9. Time taken to read synthetic data stored in various file formats against number of rows in the file.

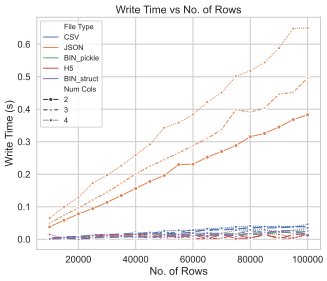

Fig. 10. Time taken to write a number of rows (No. of Rows) to various file formats.

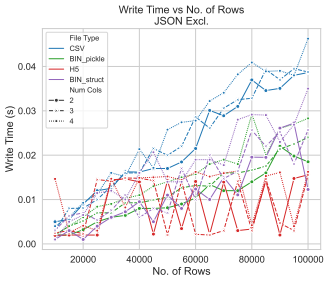

Fig. 11. Time taken to write a number of rows (No. of Rows) to various file formats without JSON.

measurements when writing to BIN with `struct` measure only the time it takes to write to the BIN files. The algorithm used during the benchmarking process has involved converting the synthetic data into raw binary but this is not taken into account in the time measurements. It was initially predicted that using `pickle` would result in longer write times due to the inherent overhead of converting Python objects into raw binary format. This process involves serialising the objects and implementing a specific structure, both of which contribute to the overall time required for writing the data. It was deduced that the background processes occurring on the DAQ computer had a significant impact, not only on the fluctuations in time shown in 11, but

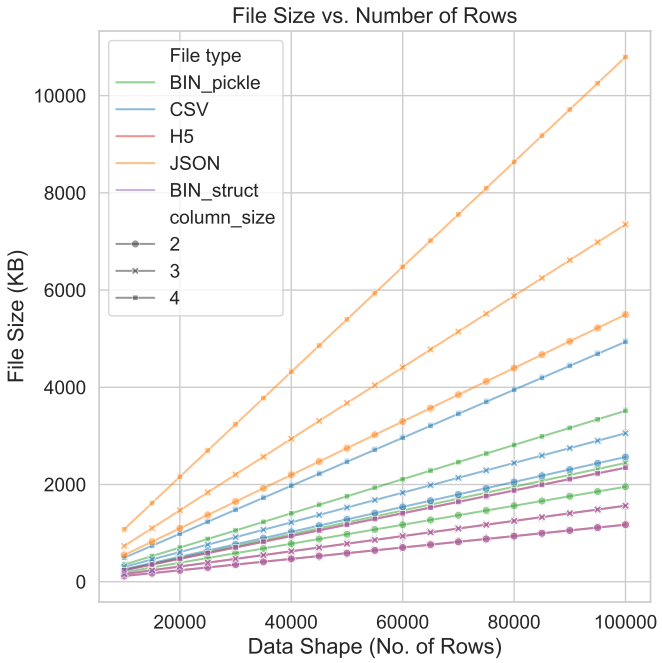

Fig. 12. File size of synthetic data generated against data shape.

also on revealing the effect of inherent serialisation overhead when writing in binary, both with and without `pickle`.

Regarding HDF5 files, figure 11 shows that not only does HDF5 write times does not scale with the size of data, but it also exhibits a regular oscillating pattern. The reason for this is due to the data structure itself implements caching when storing data [12].

Figure 12 how the file size scales with the shape of the data. In all scenarios, the file size increases linearly with the number of rows in the data. However, increasing the number of columns does not follow the same trend, especially when comparing column sizes 3 and 4. The file sizes between HDF5 and unstructured BIN were very close; with the maximum percentage difference in file size between both files was 1.7% in the worst case scenario. In general, it was found that unstructured binary files were the smallest in size. This is because they do not store any additional structure, unlike other structured file types which incorporate metadata and other organisational bits.

IV. Conclusion and Future Work

This technical report presented a variety of methods by which data could be transmitted from a Raspberry Pi Pico at the fastest rate. It was found that sending raw binary and a raw ADC reading from its temperature sensor was the method that allowed the Raspberry Pi Pico to output data at the highest rate to a DAQ computer. The average difference in timestamp was found to be $12.00 \pm 0.001\mu s$ in this scenario. Despite its advantages, this method presented several challenges. A trigger system was implemented to ensure that the DAQ computer could write the correct bits to a file. This trigger system was not perfect as $10.3 \pm 2.00\%$ of the recorded values provided non-sensical readings. Further studies within similar budgets could investigate alternative trigger systems. This could explore the impact of using larger message sizes, particularly investigating the effect of using more than one byte as the message to determine the starting bit for data writing. Buffering was also investigated but did not provide a beneficial impact on frequency. However, this could be further investigated by trying to utilise the remaining $264kB$ of RAM within the Raspberry Pi Pico rather than buffering a singular set of data. An additional avenue to increase the frequency of data transmission is to use the second core of the RP2040 to record and send readings to the DAQ computer [4].

This report also presented benchmark results for various methods of data storage. The investigated file types were CSV, JSON, BIN and HDF5. Structured and unstructured binary was also explored. The JSON file format was found to generally have the largest file size and the longest reading and writing duration. From the investigation, it can be concluded that using the JSON format so store large tabulated data is not optimal for this use case. The CSV file was the file with the second largest file size and the longest duration in the same measured quantities. Both the JSON and CSV formats are text-based and, hence, were larger in all measured quantities compared to binary-based files. HDF5 format consistently exhibited the smallest values across all measured quantities due to optimisations, such as caching, implemented by the developers of this format.

V. Appendix

The code used to carry out both investigations can be found on this GitHub repository:
https://github.com/The-Great-Nate/DAQLab_Pico_DataAnalysis

REFERENCES

[1] M. D. P. Emilio, "Data acquisition systems," *Cham, Switzerland: Springer*, 2013.

[2] J.-M. André, A. Andronidis, U. Behrens, *et al.*, "Performance of the new daq system of the cms experiment for run-2," in *2016 IEEE-NPSS Real Time Conference (RT)*, IEEE, 2016, pp. 1–4.

[3] Raspberry Pi Foundation, "Rp2040 datasheet," Raspberry Pi Foundation, Tech. Rep. 3184e62-clean, Feb. 2025, Build Date: 15 October 2024, Accessed: 5 March 2025. [Online]. Available: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf.

[4] Raspberry Pi Foundation, "Raspberry pi pico datasheet," Raspberry Pi Foundation, Tech. Rep. eec2b0c-clean, Oct. 2024, Build Date: 15 October 2024, Accessed: 5 March 2025. [Online]. Available: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf.

[5] The Pi Hut, *Raspberry pi pico*, Accessed: 2025-03-05, 2025. [Online]. Available: https://thepihut.com/products/raspberry-pi-pico?src=raspberrypi.

[6] D. G. Muratore *et al.*, "A study of successive approximation register adc architectures," 2017.

[7] cplusplus.com, *printf - C++ Reference*, Accessed: 2025-03-08, 2025. [Online]. Available: https://cplusplus.com/reference/cstdio/printf/.

[8] P. S. Foundation, *Json — json encoder and decoder*, 2025. [Online]. Available: https://docs.python.org/3/library/json.html.

[9] G. Langdale and D. Lemire, "Parsing gigabytes of json per second," *The VLDB Journal*, vol. 28, no. 6, pp. 941–960, 2019.

[10] P. S. Foundation, *Pickle — python object serialization*, version 3.13.2, Accessed: 2025-03-09, 2025. [Online]. Available: https://docs.python.org/3/library/pickle.html.

[11] T. Kurth, A. Pochinsky, A. Sarje, S. Syritsyn, and A. Walker-Loud, "High-performance i/o: Hdf5 for lattice qcd," *arXiv preprint arXiv:1501.06992*, 2015.

[12] T. H. G. John Readey, "Improve hdf5 performance using caching," 2022. [Online]. Available: https://www.hdfgroup.org/2022/10/17/improve-hdf5-performance-using-caching/.