DinoGo: An Autonomous (ML) Dinosaur Runner Game

Zain Mughal
Department of Physics, Imperial College

Abstract—**This report details the design, prototype evolution, optimisation, testing, and evaluation of *DinoGo*, an embedded endless runner game developed entirely in low-level assembly on a 8-bit PIC18F87K22 microcontroller, with later advancements featuring a simple 3–8–3 feedforward neural network for autonomous gameplay. To overcome the tight timing constraints imposed by the 128×64 GLCD, a novel partial screen redraw strategy was later implemented, achieving a 8.7× speedup over the initial full screen clearing method, this allowed for a consistent screen response rate of 191 FPS ($\pm$0.017 FPS), while full-screen clears were measured at 300–410 ms ($\pm$4.17 ms) per clear ($\sim$2-3 FPS). All metrics were evaluated using high-speed video recordings (240 FPS), yielding a button input latency between 33.3–54.2 ms ($\pm$4.17 ms), and real-time ML inference time of 34.1 ms ($\pm$4.17 ms) per execution. Tactical memory management measures restricted RAM usage to a maximum of 15.1% of the available 4 KB, avoiding the need for external memory despite supporting 56 ML parameters. Testing across gameplay configurations confirmed that system bottlenecks resulted from GLCD hardware alone, not MCU computation, and that smooth gameplay was preserved under load. Therefore, our results validate the feasibility of demanding graphical games and real-time embedded AI on ultra-low-resource platforms and motivate future improvements in display hardware and on-chip neural architectures.**

I. Introduction

Microcontrollers are low-power, cost-effective computing platforms that have long been the backbone of embedded systems in applications ranging from household appliances to industrial controls [1]. Despite their limited resources, modern microcontroller units (MCU) can execute surprisingly complex tasks when paired with efficient algorithms and optimised code. The recent surge in embedded artificial intelligence has further expanded their utility, enabling on-device decision-making in real time [2].

The *Chrome Dino* game is a simple, an endless runner featuring an adorable jumping T-Rex that appears in the Google Chrome browser when connectivity is lost, showcasing a minimalist game design that manages to engage users with its straightforward mechanics and simple yet charming retro aesthetics [3]. Inspired by this, our project, *DinoGo*, aims to re-create the game on an 8-bit MCU while extending its functionality through embedded machine learning (ML). Our primary targets, outlined in our proposal, were achieving a playable frame rate of 30–60 FPS (redefined to *response rate* to reflect actual readings), maintaining input latency near 20 ms, and sustaining real-time ML inference below 20 ms. These metrics underpinned our design decisions and guided our process of iterative design refinement, optimisation trials, and careful system evolution, all to enable optimal prototype evolution from an initial concept to a final minimal viable product (MVP).

Initially, our efforts focused on developing a fully playable game engine that achieves smooth sprite animation and responsive user controls, despite the severe constraints of the PIC18F87K22 (which offers only 4 kB of RAM and 128 kB of Flash [1]). A 128×64 Graphical Liquid Crystal Display (GLCD) was chosen for its ease of use, availability and parallel interface; however, its inherent (worst-case) slow refresh rates forced careful timing and clever screen redrawing techniques to maintain a stable frame/response rates despite expanding game demands.

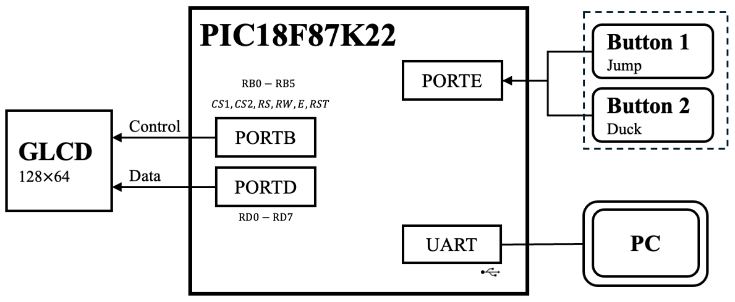

Fig. 1. High-level hardware block diagram for the DinoGo system. The PIC18F87K22 is centrally located on the EasyPic PRO v7 development board and interfaced with a 128×64 GLCD via an 8-bit parallel bus (control signals: RS, R/W, E, CS). Two tactile buttons (Jump, Duck) enable user input, while a UART port is employed for data logging and debugging. This diagram is a simplified version from the original proposal document.

Once the game engine was robust, a compact feed-forward neural network, configured with a 3–8–3 architecture and later quantized down a to 8 bit parameter grid, was integrated to perform autonomous gameplay. This ML module was implemented using multi-byte (Q8.8) fixed-point arithmetic, ensuring that its inference time remained within the stringent timing constraints of the game loop whilst maintaining scope complexity [5].

Although we implemented a complex and robust 16-bit random number generator (RNG) based on a 16-bit Linear Feedback Shift Register (LFSR) using the polynomial $x^{16} + x^{14} + x^{13} + x^{11} + 1$, yielding a maximal period of 65,535 and produces a near-uniform distribution of pseudo-random numbers (see figure 5). In our system, the RNG injects variability by generating different seeds for each spawning event, thereby preventing any discernible pattern that might allow the ML model to overfit to a specific sequence. However, as the neural network was trained off-chip, its performance remains independent of the RNG's output or hardware implementation regarding training. Consequently, while the RNG plays a supporting role in maintaining game unpredictability and experience, it is not a core focus of our evaluation, but a welcomed positive result nonetheless.

The motivation for this project lies in demonstrating that an 8-bit microcontroller can handle both dynamic, resource-intensive graphics and embedded ML simultaneously. This achievement not only reinforces the potential of cost-effective, real-time systems, but also opens the door for future applications in robotics, automotive systems, and portable consumer electronics, where robust advanced analytics must be performed on a constrained hardware budget.

This approach goes well beyond the typical scope of microprocessor lab exercises and, to our knowledge, no prior 8-bit implementation of a neural-agent Dino runner exists, demonstrating a novel application of constrained embedded AI.

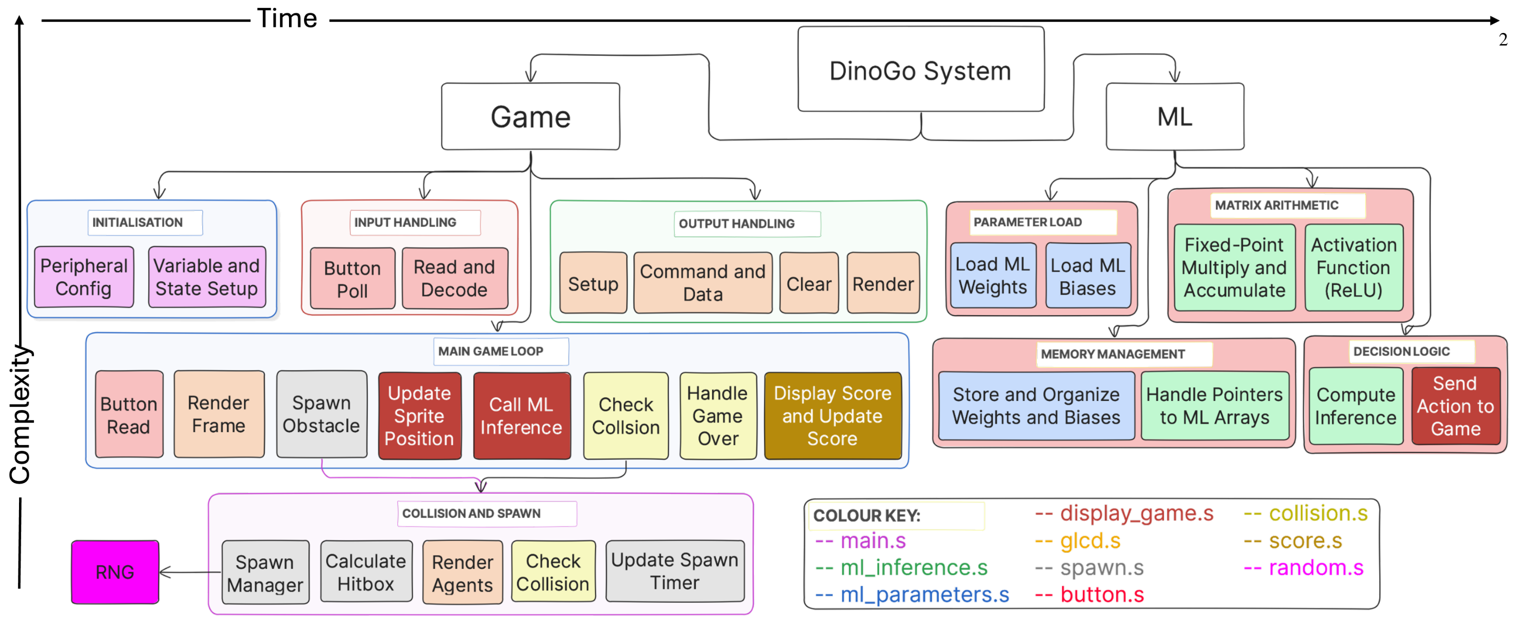

Fig. 2. Top-down modular diagram of the *DinoGo* embedded software system. The architecture is divided into two principal subsystems: *Game* (left) and *ML* (right), each comprising multiple functional modules. Execution flows broadly from left to right (initialisation to runtime decision logic), whilst top to bottom is in ordered of less complicated sub-routines. Color-coded blocks correspond to individual source files as labelled in the 'colour key', improving codebase traceability and modular maintainability. (Note: The colors of the encapsulating translucent blocks e.g. `INITALISATION` have no meaning, simply viusal separators.)

II. Design and Methodology

A. High-Level Design

The overall physical architecture of *DinoGo* is organised into two main subsystems: the hardware platform and the software architecture. At its core, the project was developed in a sequential manner, first establishing a fully playable, real-time game engine and subsequently integrating an embedded machine learning (ML) module. This section outlines the system's structural framework, emphasising the interplay between hardware constraints and software optimisations, that enable both smooth game execution and autonomous decision-making.

Figure 1 presents the high-level hardware block diagram, which shows the PIC18F87K22 microcontroller at the center. The MCU interfaces with the GLCD via an 8-bit parallel (data & control) bus (with control signals RS, R/W, E, CS1, CS2, and RST), two tactile input buttons (for jump and duck functions). The chip sits on the EasyPIC PRO v7 development board and is connected to a range of auxiliary peripherals such as a UART port for data logging and debugging, multiple timers, PWM channels among other interfaces, but most importantly, an in-circuit debugger via the PicKit4 system allowed for robust connectivity and direct debugging. The diagram allows us to visualise the critical hardware specifications, namely the limited RAM and Flash, as well as the GLCD's sequential refresh limitations, which ultimately govern the perceived real-time performance of the system, despite the MCU being able to run at up to 64MHz [1][4].

Complementing the hardware view, Figure 2 displays the top-down modularity diagram of the software architecture. This diagram partitions the system into its two major sub-systems, the game itself and the ML. Each module/sub-routine is designed to address specific challenges, such as handling user input, managing real-time display updates, or executing fixed-point neural network calculations, while operating within the tight resource limits imposed by the PIC18F87K22. The hierarchical structure illustrated here serves as a roadmap for efficient code organisation and facilitates debugging and future system enhancements.

The real-time operation of *DinoGo* is governed by the Main Game Loop, as shown in Figure 3. In each iteration of this loop ($\sim$ game/update tick), the system polls for user input (using a polling routine to avoid the compute overhead of interrupts as well as need for any complex debouncing routines), updates the dinosaur's state (transitioning between running, jumping, descending, and ducking), and spawns obstacles using a 16-bit LFSR based algorithm. To optimise performance given the slow refresh rate of the GLCD, only portions of the screen that require updating are redrawn, thus minimising instruction cycles per game tick. This design ensures that the game maintains a high and stable screen response rate, even under varying rendering loads.

B. Hardware Overview

The hardware platform for *DinoGo* is built around the PIC18F87K22 MCU, which, despite its limited memory, provides sufficient computational power and peripheral support to execute a real-time game engine with later integrated machine learning. Given these resource constraints, our design required rigorous optimisation in both memory allocation and processing efficiency, ensuring that game logic and ML inference operates reliably within the available computational envelope.

A central element of the system is the 128$\times$64 GLCD, that serves as the primary visual output device, interfaced with the MCU via a (hard-wired PCB) 8-bit parallel bus using dedicated control lines to coordinate data transfers, although, the GLCD is inherently limited by slow refresh rates; its response times are approximately 200–300 ms for rising and 150–200 ms for falling edges, as specified in the manufactures datasheet [4]. To mitigate these limitations and avoid any visible flicker, our design employs partial screen updates rather than full-screen redraws on each frame update, as well as custom delays (5ms tick with 1ms delay) and timing so that our data processing does not overwhelm the GLCD by outputting at a rate greater than the limited display technology can handle. This

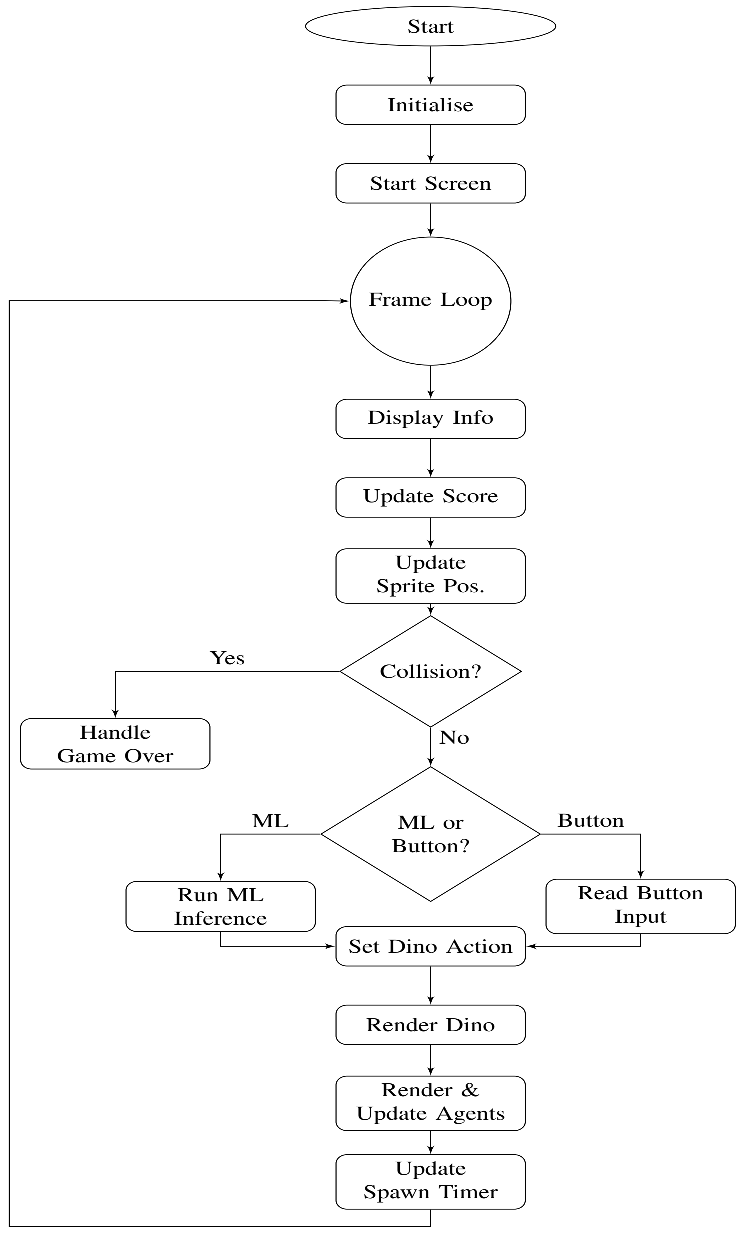

Fig. 3. Main game loop flowchart for DinoGo. Each iteration (or *game tick*) runs in approximately 5 ms of real time, including an inbuilt 1 ms delay and proceeds through initialisation of the screen, periodic updates to display info, score, and sprite positions, followed by a collision check. If a collision occurs, a *Game Over* routine is triggered; otherwise, control passes to either an ML inference path or button-driven logic to set the dinosaur's action. Finally, the system renders the dinosaur, updates obstacle agents, and increments the spawn timer before looping back to the top. This structure ensures real-time responsiveness on a constrained 8-bit platform with partial screen refreshes and fixed-point arithmetic.

approach not only conserves processing time but also preserves the visual integrity of the display under dynamic gaming conditions.

User input is provided by two tactile buttons that allow the player to command the dinosaur to jump or duck. Instead of using interrupts, which can be susceptible to noise and require additional debouncing logic, the system polls the buttons once per game tick. This method greatly simplifies our design and reduces computational overhead, ensuring that user input is reliably captured without sacrificing system performance.

Auxiliary peripherals, such as the UART port, contribute to data logging and debugging, this channel was especially useful during earlier development for monitoring button input stages and verifying GLCD rendering routines. Although earlier iterations of the design considered the integration of a PWM-driven buzzer and an external Electrically Erasable Programmable Read-Only Memory (EEPROM) for storing ML parameters, these components were ultimately omitted from the final build due to trivial priority/impact and time constraints. The decision against the latter was based on the efficient memory management achieved in our initial prototype with the on-chip resources, which remained below a maximum of 15% utilisation even after ML integration.

Additionally, due to the PIC18F87K22's Harvard architecture, our design had to carefully manage the allocation of variables. Frequently accessed data (such as ML parameters and game state variables) were assigned to the access bank to minimise the number of bank-switching cycles, which can otherwise increase processing time, as 2 cycles would be required instead of 1. Less frequently used variables were pushed over to the banked RAM memory, making them slower to read (fastest 1 cycle $\sim 15.6ns \ll$ display refresh).

Figure 1 serves as a visual summary of the hardware architecture that underpins the entire *DinoGo* project.

C. Software Overview

The software architecture of *DinoGo* is designed with a strong emphasis on modularity and real-time efficiency. The system is implemented entirely in low-level PIC18 assembly code and works to exploit device architecture to maximise performance. The design strategy was to first establish a robust, fully functional game engine and then at a later stage, to integrate a compact ML module for autonomous decision-making and gameplay. This section describes the key modules, the flow of operations during each game tick, and the design choices made to ensure real-time responsiveness.

At the core of the software is the *Main Game Loop*, which centrally orchestrates all game activities, with each `game tick` representing one complete iteration of input handling, state updates, rendering, and ML inference, executing at a constant duration of $5.14 \pm 0.14\,ms$ (confirmed via testing), including an intentional inbuilt 1 ms delay to regulate output to the GLCD.

The `game_speed` parameter governs how often obstacle positions are updated. Specifically, obstacles move leftward by 8 pixels every X game ticks, effectively defined as: $\texttt{game\_speed} = 21 - X$. Initially, $X = 20$, resulting in slow obstacle movement, and X gradually decreases to 1 as the game progresses, leading to a continuous increase in perceived speed.

Separately, we define the *response rate* as the number of partial screen refreshes per second, which better captures the visual update performance on the constrained GLCD hardware. At maximum speeds, the `frame tick` frequency approaches the `game tick` frequency, maximising visual responsiveness.

Input Polling: The system polls the tactile buttons for user input (jump or duck). Polling is preferred over interrupts to avoid spurious triggers from button bouncing and to simplify the control logic. The short duration of each game tick ensures that polling remains consistently responsive to user actions.

State Update: Based on the input, the current state of the dinosaur is updated (transitioning among running, jumping, descending, and ducking) and rendered accordingly.

Obstacle Spawning: A 16-bit polynomial enabled LFSR generates pseudo-random numbers to determine when and what type of obstacle to spawn (e.g., small/medium/large cacti or bird) and is called once every 50 game ticks. The spawn routine and RNG is optimised for minimal computational overhead and uniform randomness (Figure 5). Due to display and timing constraints, only two obstacles can be shown on-screen simultaneously; thus, we limit to two obstacle spawning agents with capabilities to expand this. See appendix figure 7 for the respective logic flowchart.

Score and Collision Checks: The multi-byte score is incremented based on cumulative frame ticks, ramping from 0 to 999 based on survival time and updating every 30 game ticks. Collision detection is performed via checking overlap between defined hit-boxes per sprite, executed each game tick. If a collision is detected, the game transitions to a game-over routine, which saves the high score (if `current_score` > `saved_high_score`), triggers the `game_over` flag and initiates the game over sequence and screen.

ML Inference: The `ML_OPTION` flag coordinates between using the ML module or reading the button input directly (manual mode) to decide the dinosaur's next action. The ML module collects the state vectors, loads quantized parameters, computes the hidden and output layers using (Q8.8) fixed-point arithmetic combined with the ReLU activation function implemented in assembly across all 56 network parameters (*feed forward 3–8–3 network* $\implies 3 \times 8 = 24$ *weights and* 8 *biases in the input-to-hidden layer, plus* $8 \times 3 = 24$ *weights in the hidden-to-output layer, giving a total of* $24+8+24 = 56$ *parameters)*, and outputs a decision/action (jump, duck, or idle). We chose 3 inputs (obstacle distance, obstacle type/height, and dinosaur state) to minimise parameters while retaining essential game features. This compressed the model from 14 potential dimensions to just 3 dimensions using clever feature engineering, which fit neatly into the MCU's limited access bank without causing excessive bank switching. The ML inference is designed to run in real-time (typically under 20 ms), ensuring it does not extend the game tick beyond acceptable limits and lag the compute.

Display Updates & Rendering: The system updates the GLCD by redrawing only the portions of the screen that have changed (partial screen updates) to overcome the slow refresh rates of the display, we do this by tracking the obstacle spawn agents as they move and render as well as the Dino [7]. This is particularly important because clearing the entire screen in each tick would significantly extend the information for the GLCD to process and slow it down significantly. Finally, the dinosaur and all active obstacles (agents) are rendered on the GLCD via looping through their bitmap pixels byte-by-byte or bit-by-bit for tricky animations (the flying bird), then the spawn timer is updated. A curved loop-back then re-initiates the next game tick. The split boundary in the GLCD (CS1 & CS2 - Left and Right) is handled by displaying portions of the bitmaps on each side till the boundary has been seamlessly crossed.

The duration of each game tick is fixed at approximately 5 ms, including an inbuilt delay of 1 ms. This ensures that updates remain within GLCD timing limits and preserves stable visual output across all configurations. It is important to note that while these delays are programmed into the loop to ensure correct display timing and to prevent display flicker, the exact duration of a tick must be measured manually during testing to account for any overhead. This design decision is critical: if extensive operations, such as multiple full-screen clears, were executed in one tick, the game performance would degrade significantly, thereby impacting real-time responsiveness.

The overall software is modularised into distinct components (as seen in figure 2), that together implement the game's functionality on the PIC18F87K22. The main entry point, defined in `main.s`, initialises the hardware, configures the system and timing routines (with additional configuration specified in `config.s` and `delay.s`), and drives the continuous game loop (refer to Figure 3). Within each tick, the loop updates the game state by processing user inputs through routines in `Button.s` (which handles polling and decoding), updating sprite positions, managing collisions via `collision.s`, and spawning obstacles using a pseudo-random generator implemented in `spawn.s` and `random.s`. Display updates are efficiently handled by the `GLCD.s` and `Display_Game.s` modules, which employ the partial screen techniques. The ML Inference module, coordinated by `ml_handler.s` and executed in `ml_inference.s`, implements and executes the simple neural network logic to feed actions back into the game loop. Additionally, `score.s` manages (16-bit using overflows and most significant tracking) score tracking and high-score updates, while careful manual memory management throughout ensures that game state data and ML parameters can coexist within the limited bank and RAM memory.

III. Results and Discussion

This section presents the experimental data gathered during testing and provides an analysis of the system's performance relative to the design objectives. Both quantitative measurements and qualitative observations are discussed, leading onto potential improvements for future iterations.

A. Quantitative Results

To critically evaluate the performance of *DinoGo*, we quantitatively measured several key metrics across configurations. Data was collected over five independent 60-second game-play sessions, each repeated twice, using mostly high-speed visual analysis via an iPhone 16 Pro recording at 240 ±1 FPS, propagated standard deviation uncertainties were calculated to capture run-to-run variability.

Screen Response Rate: This was initially measured using UART timestamped logging. However, this method introduced a slowdown factor of 43×, reducing the observed rate from 191 FPS to 4.4 FPS, rendering it unsuitable for realistic analysis. We pinpoint this effect due to the multiple requests required from both the processor and GLCD to output data simultaneously every game tick. We therefore adopted for analysis via high-speed visual recordings, which better reflect user-perceived performance. Final response rate measurements were obtained under 2-agent manual configurations at varying game speeds, from 1 (fastest) to 20 (slowest), using manual analysis frame differencing between a LED indicator and a display response. The system achieved a sustained average response rate of 191 ±1 FPS (or ±4.17 ms), consistent across repeated trials, thus successfully overshooting our target of 30–60 FPS. This metric represents the frequency of partial screen updates, which avoids full contrast inversion and provides a more accurate indicator of visual performance under real gameplay conditions. Measurement uncertainty was based on frame sampling granularity and negligible human alignment error.

Input Latency: Button input latency was measured using high-speed footage to capture the delay between a button press/release and a visible on-screen response. Across trials, latency ranged from 33.3 ms to 54.2 ms (±4.17 ms). The discrepancy between button press and release latencies is currently unexplained but likely attributable to GLCD rendering behaviour rather than signal propagation. Further investigation with GPIO toggling and logic analysis was not feasible due to project time limitations. Although latency performed worse than initially aimed for, the resulting responsiveness remained within acceptable perceptual bounds for gameplay.

ML Inference Time: ML inference is triggered on obstacle spawn, not every tick, to reduce overhead. The average measured by triggering an LED flash when the input vector was ready, and measuring the delay until the dinosaur visibly responded on screen. This resulted in an inference time of 34.1 ms ± 4.17 ms, significantly exceeding the original 20 ms target. However, this value includes not only the compute time but also GLCD rendering, memory access overhead, and bank switching latency. Given the 5 ms game tick duration and GLCD-bound visual delays (300–410 ms), this inference time remains imperceptible in context and does not degrade gameplay. Moreover, this timing reflects worst-case overhead and is likely an upper-bound estimte.

RAM Usage: Memory utilisation was extracted from MPLABX linker map files and build simulations. The base game used $\sim$ 10.2% (420 bytes) of available RAM. After ML integration, this rose to $\sim$ 15.1% (620 bytes), accounting for network weights, intermediate buffers, and runtime variables. No external EEPROM was required, and all critical variables were prioritised to the access bank, whilst lower priority variables were allocated to regular banks, therefore reducing instruction overhead due to bank switching. Due to the overall complexity of our codebase we exceeded the maximal variable allocations to the access bank ($2^8 - 1 = 255$) and experienced COMRAM issues, forcing us to manually be deliberate with our allocations. Any further iteration of this project would require careful banked memory management beyond our methods.

GLCD Clearing Time and Optimisation: We benchmarked full-screen clearing versus partial updates to quantify GLCD bottlenecks. Using high-speed camera footage over 1000 frames, full-screen clearing was found to take approximately 2.64 s per frame in worst-case conditions, with high non-quantifiable uncertainty due to flicker-based human interpretation. To address this, partial localised clearing was implemented, improving update speed to between 300–410 ms depending on game speed (game speed 1 = slowest and 20 = fastest respectively). These values match the expected nonlinear GLCD behaviour described in the datasheet (rise/fall times up to 300 ms), and are visualised in Figure 4. This optimisation yielded an 8.7× speedup in practical rendering, reinforcing the GLCD as the primary bottleneck post optimisations.

Measured System Performance Metrics

Metric	Measured Value	Target / Reference
Screen Response Rate (2-Agent)	191 $\pm$ 1 FPS	30–60 FPS
GLCD Clearing Time	300–410 $\pm$ 4.17 ms	Minimal
Input Latency Range	33.3–54.2 $\pm$ 4.17 ms	$\sim$20 ms (ideal)
ML Inference Time	34.1 $\pm$ 4.17 ms	$<$20 ms
RAM Usage (Game Only)	10.2% of 4 KB	$<$100%
RAM Usage (With ML)	15.1% of 4 KB	$<$100%
UART Slowdown Factor	43×	–
Partial-to-Full Render Speedup	8.7×	–

Table 1: Summary of measured system performance metrics. The screen response rate refers to the number of partial screen refreshes per second during 2-agent gameplay. Full-screen clearing refers to full contrast inversion times on the GLCD. Input latency values correspond to different button-triggered events. All uncertainties were derived from high-speed camera recordings (240 FPS), using standard deviation and frame-resolution bounds (±4.17 ms or ±1 FPS). Game tick duration was measured at 5 ms, with an in-built code delay of 1 ms.

B. Qualitative Observations

Informal subjective testing and visual analysis of our alpha prototype offered insights into real-time responsiveness, display behavior,

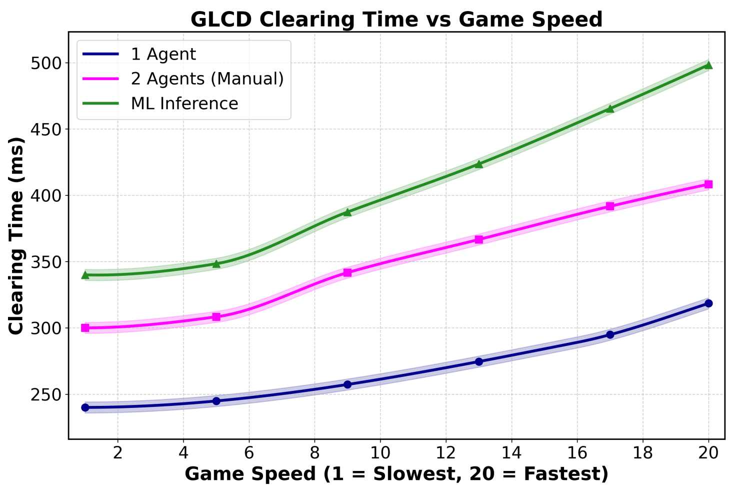

Fig. 4. Measured GLCD clearing time vs game speed (1 = slowest, 20 = fastest) across three control configurations: 1-agent (manual, baseline), 2-agent (manual), and ML-enabled. Clearing time increases non-linearly with game speed and agent complexity due to higher GLCD data throughput. The ML curve shows a consistent offset above the 2-agent configuration due to inference overhead. Shaded regions represent a constant timing uncertainty of ±4.17 ms (1 frame at 240 FPS), reflecting camera resolution limits. These results highlight the compounded visual latency introduced by agent count and embedded compute, reinforcing the GLCD as the system's dominant bottleneck.

and ML control effectiveness. Gameplay was observed across all configurations (manual dual/single agent and ML), live test runs conducted under typical operating conditions among a sample of $\sim$5 alpha testers.

The GLCD, though limited in refresh characteristics, produced sufficiently stable visual output throughout. Our use of partial-screen updates effectively suppressed flickering and ghosting in all low to medium-speed scenarios. At higher speeds and obstacle densities, minor visual artefacts were observed, particularly ghost trails when jumping or incomplete screen wipes during full-frame clears, but these were mostly mitigated under our optimised partial redraw system. Frame stability remained sufficient for visual clarity, and sprite transitions were reported by alpha test users as *"smooth and natural"*.

Input responsiveness was validated both by user feedback and high-speed footage. Button-triggered actions such as jumping and ducking appeared on-screen with negligible perceptual delay, consistent with our latency measurements (33.3–54.2 ms ± 4.17 ms). Users described the gameplay as *"responsive"* and *"fluid"*, with no indication of input lag affecting their experience within low-medium game speeds.

The ML-controlled mode demonstrated reliable obstacle response in all standard gameplay cases. During real-time inference, actions triggered by the neural network matched the expected decision pattern from the Python-trained baseline. Although the model does not manage to play the game very well, this is an issue of the model training itself, and performs exactly the same on an unrestricted Mac python environment, thus the hardware and run systems do not interfere with or impact the ML. Despite this, we can still still see a very jump biased model successfully jump over obstacles and survive. These moments reinforce the feasibility of real-time ML inference on constrained systems.

The ML system's general behavior was stable, visually explainable, and deterministic. Users noted that the dinosaur acted *"instinc-*

tively jumpy" in response to threats as expected, and did not exhibit any unexpected jittery or erratic behavior. Despite being compressed from a full 14-dimensional, 9667-parameter (14–64–128–3) model down to 3 input features and 56 total parameters, the network maintained its decision integrity under load. This consistency, coupled with inference times under 34.1 ms ± 4.17 ms and minimal RAM overhead, validates the integration of embedded AI into 8-bit architectures without loss of user experience.

Together, these qualitative insights support the quantitative data and the narrative that *DinoGo* delivers a playable, stable, and perceptibly intelligent experience, even under the constraints of a legacy microcontroller environment.

IV. Potential Updates, Modifications and Improvements

The primary performance bottleneck consistently identified in the current *DinoGo* system is the GLCD. Although the PIC18F87K22 MCU, based on Harvard architecture, is capable of rapid instruction execution and efficient data handling, its limited RAM and particularly the restricted access bank memory impose significant challenges. In our current implementation, we experienced difficulties with access bank memory allocation exceeding 255 variables, leading to conflicts in COMRAM usage. Future revisions could benefit from a more sophisticated memory management strategy, such as automatically reassigning less frequently accessed variables to standard banks or employing overlay memory techniques with stricter timing controls to prevent unintended data overwrites (techniques partially implemented currently). The current MCU bank limitation puts an upper limit on the amount of maximal ML parameters and variables we could practically employ.

Beyond memory management, the GLCD itself remains the chief obstacle to higher performance, despite multiple careful optimisation iterations, like partial-screen updates, the inherent slow refresh rates and ghosting issues limit the overall screen response rate and perceived visual clarity, regardless of the compute efficiency behind it. Therefore, upgrading to a display technology with faster response times or a more advanced graphics interface would significantly alleviate this bottleneck. Additionally, although the current ML module performs inference within acceptable timing constraints given the display latency, there remains potential to explore a more complex and optimised model; a better trained and performing model would also improve the actual autonomous gameplay, however this is not a direct evaluation criteria in our scope. A network with higher dimensional inputs, potentially reverting to a more accurate 14-dimensional configuration (even if compressed to fit available resources), could improve decision quality, provided that memory and processing overheads are adequately managed. This would also allow us to effectively investigate the relationship between model complexity and inference times on a restricted embedded system.

Moreover, the precision of input timing measurements and overall system diagnostics can be greatly enhanced by integrating a more sophisticated data logging framework. For instance, employing dedicated digital data loggers or logic analysers to monitor GLCD data bus utilisation could reveal transient bottlenecks, enabling finer calibration of the system's timing parameters. Finally, upgrading to a MCU with a more advanced memory architecture, or an alternative architecture entirely (e.g., STM32F4 or STM32H7 [8][9]), could significantly reduce access bank contention and simplify memory management. This would enable more variable states to reside in fast-access memory, facilitating more complex ML implementations without performance penalties from bank switching. A dedicated or expanded UART buffer would also help to eliminate the 43× slowdown observed during data logging, allowing direct data capture without disrupting gameplay and enabling more detailed, real-time performance analysis.

V. Conclusion

In this report, we have presented *DinoGo*, an entertaining autonomous runner game developed on the PIC18F87K22 microcontroller that integrates a compact 3–8–3 feed-forward neural network for real-time autonomous gameplay. The project progressed through two major development stages: first, the construction of a fully playable, graphics-based game engine in low-level assembly, and second, the integration of a fixed-point ML inference module to enable autonomous control. Each phase presented unique technical challenges, from managing limited memory and instruction throughput to achieving frame-stable visual rendering on a 128×64 GLCD. The final MVP demonstrates that both stages were successfully realised, first by establishing a responsive, visually rich game loop within the tight timing constraints of an 8-bit system, and second by deploying an embedded AI agent that performs real-time decision-making without external processing or memory support.

Quantitative and qualitative results confirm that the final prototype performs within our design specifications: an average screen response rate of 191 ± 1 FPS, input latency of 33.3-54.2 ± 4.17 ms, and ML inference completed within 34.1 ± 4.17 ms, although this exceeds our original *sub 20 ms* target, it remains imperceptible in practice due to GLCD refresh delays and does not degrade gameplay responsiveness. Memory usage remained below 15.1% of on-chip RAM, with no reliance on EEPROM or auxiliary storage. These results validate the feasibility of executing both resource-intensive graphical operations and ML inference concurrently on ultra-constrained embedded hardware.

The primary limitation of the current system remains the GLCD, whose inherent refresh delays and ghosting restrict the achievable response rate and visual clarity, particularly at higher game speeds. Although extensive optimisation, particularly partial-screen redraws greatly mitigates this bottleneck, it remains the principal constraint on display fidelity and responsiveness at higher game speeds.

Nevertheless, *DinoGo* showcases the versatility and untapped potential of classical microcontrollers to perform tasks far beyond simple control logic. The successful development of a fully functional graphical game, followed by the integration of lightweight, real-time ML on-device, highlights that 8-bit platforms can support both rich user experiences and intelligent autonomy. These findings challenge the notion that embedded AI requires expensive specialised hardware, instead proposing a design philosophy where performance is extracted through careful scheduling, model compression, and system-level optimisation. Such designs could serve a growing class of edge devices in safety-critical, latency-sensitive environments, from smart automation to robotics, where robust, deterministic and energy-efficient systems are essential. As an MVP proof-of-concept, *DinoGo* affirms the viability of deploying responsive, AI-driven functionality on legacy microprocessor platforms, opening doors for further innovation in embedded intelligence. Future iterations could extend this approach on modern architectures (e.g., ARM Cortex-M) to enable even more sophisticated embedded autonomy.

References

[1] Microchip Technology Inc., *PIC18F87K22 8-Bit Microcontroller Data Sheet*, Revision A, Document DS30009960, Available: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/PIC18F87K22-Family-Data-Sheet-DS30009960.pdf

[2] Sutton, R. S. and Barto, A. G., *Reinforcement Learning: An Introduction*, MIT Press, 2014, 2015.

[3] Google Chrome Dinosaur Game, Available: https://chromedino.com.

[4] EastRising (BuyDisplay), *128×64 Graphic LCD Module Data Sheet (ER-OLED-009)*, Accessed via Scribd, Available: https://www.scribd.com/document/496586052/Glcd-128x64-Spec

[5] Mnih, V., et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] Bolanakis, D., Laopoulos, T., & Kotsis, K. (2016). "Fixed-point Arithmetic for a Microcomputer Architecture Course," *IEEE Tech. and Engineering Education*, 9, 1–7.

[7] Dash, S. (2016). "Graphical LCD Device Driver Development and Optimization of Boot Time for Embedded Device," *PhD Thesis*, RG Publication. DOI: 10.13140/RG.2.2.33575.75687

[8] STMicroelectronics, *STM32F4 Series: High-performance MCUs with DSP and FPU instructions*, Revision 6, Datasheet, 2022. Available: https://www.digikey.co.uk/en/htmldatasheets/production/880529/0/0/1/stm32f407vgt6

[9] STMicroelectronics, *STM32H7 Series: High-performance and DSP MCUs with up to 550 MHz CPU*, Product Overview, 2023. Available: https://www.st.com/resource/en/datasheet/stm32h743vi.pdf

Acknowledgements

I acknowledge the use of ChatGPT-4o (OpenAI, https://chat.openai.com/) to assist in refining the grammar and structure of this report. This tool was not used for generating content or performing any analysis; it served solely as a language and presentation assistant. I confirm that the content and analysis presented are entirely my own.

Additionally, I acknowledge the use of ChatGPT-4o during the lab sessions as a supplementary tool, functioning more as a "virtual demonstrator" to help with code debugging, interpretation of documentation, and understanding of nuanced microcontroller behavior. Parts of the code base were developed with the assistance of this tool, under my direct guidance and supervision.

I am also grateful for the invaluable support and guidance provided by our demonstrators throughout the lab. Special thanks to *Zhonghe Feng & Mark Neil*.

Finally, I remain grateful for the collaboration and invaluable contributions of my lab partner throughout this project, *Yixui Yin*.

Appendix A: RNG Distribution Plot

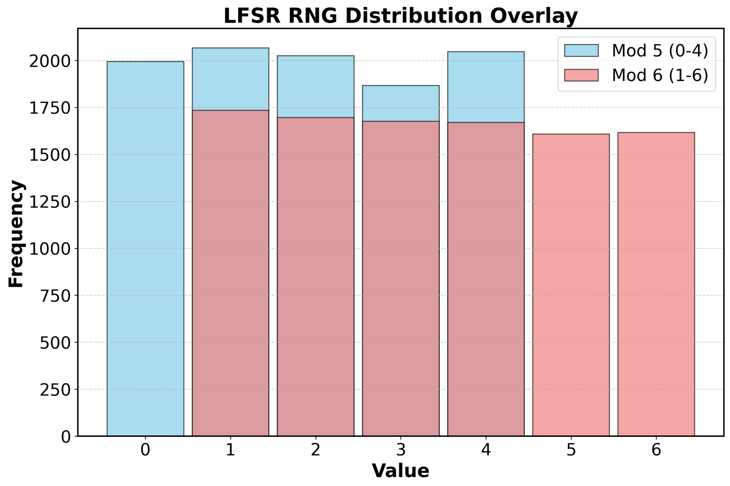

Fig. 5. Histogram overlay showing frequency distributions for LFSR generated random numbers. Modulo 5 (blue, values 0–4) and modulo 6 + 1 (red, values 1–6). The mod 5 distribution is near-uniform, while the mod 6 + 1 distribution shows slight bias towards lower values, due to imperfect divisibility and post-processing limitations.

Appendix B: GitHub Repository

The complete codebase repo and additional documentation for this project are available on GitHub:

- https://github.com/IssueIN/DinoGo/tree/prototype

Appendix c: Dinosaur Action Logic

Fig. 6 illustrates the finite state machine (FSM) governing dinosaur behavior in the game. The system responds to button inputs or ML output to transition between states: Running, Jumping, and Ducking. Each transition includes constraints (e.g., no mid-jump input override), ensuring animation completeness and gameplay consistency.

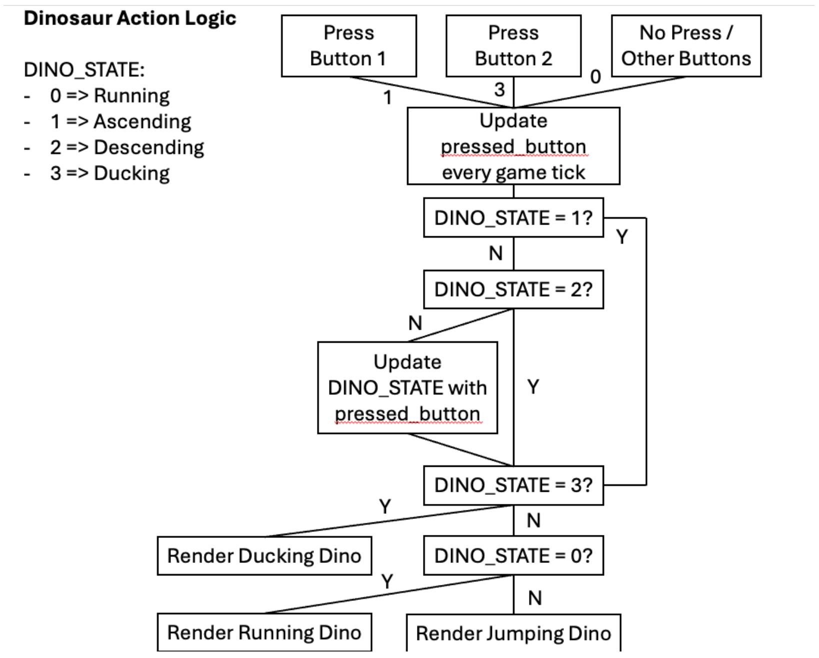

Fig. 6. Finite state machine diagram for dinosaur action logic.

Appendix D: Obstacle Spawning Logic

Fig. 7 shows the logic used to spawn and update obstacles during gameplay. Two agents are deployed to manage concurrent obstacles, with minimum spacing constraints to ensure fair timing for player response. Obstacle types and properties are selected via LFSR-based random number generation.

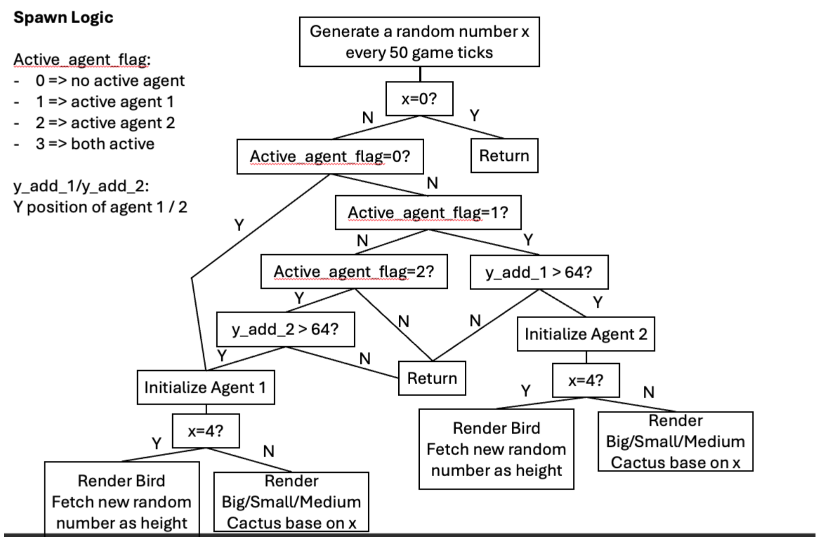

Fig. 7. Logic diagram for obstacle spawning and management.