# PROVISIONAL PATENT APPLICATION

**Title:** Systems and Methods for Grammar-Constrained Code Generation and Execution on Resource-Limited Embedded Devices

---

## CROSS-REFERENCE TO RELATED APPLICATIONS

None.

---

## FIELD OF THE INVENTION

This invention relates to embedded systems, code generation, machine learning, domain-specific languages (DSLs), and hardware/software co-design. It concerns systems that convert natural-language descriptions of device behavior into verified executable control programs suitable for microcontrollers and other resource-constrained computing platforms.

---

## BACKGROUND OF THE INVENTION

Microcontrollers used in consumer electronics, robotics, IoT, and industrial systems remain difficult to program. Developers must work in low-level languages such as C and C++, manage memory manually, and reason about timing, interrupts, and device-specific registers. This complexity prevents rapid prototyping and excludes non-expert users.

Large language models demonstrate the ability to generate source code from natural language descriptions, but such models are computationally intensive, require GPU-class hardware, and often produce syntactically invalid or unsafe code. They cannot run on microcontrollers due to memory and performance constraints.

Simultaneously, existing high-level interpreters for MCUs—such as MicroPython or JavaScript variants—consume hundreds of kilobytes of RAM and introduce unpredictable timing behavior. Other tools, such as block-based systems, rely on cloud compilation or fixed templates rather than true code generation.

No system currently provides:

1. natural-language-to-DSL generation,
2. syntactic correctness guarantees at generation time,
3. efficient machine-learned model adaptation for embedded semantics,
4. translation to a deterministic bytecode executable on small MCUs, and
5. a verified micro-interpreter with bounded memory and predictable execution.

This invention fills this gap.

---

## SUMMARY OF THE INVENTION

The invention provides a system that accepts natural-language input describing desired behavior of an embedded device and produces a syntactically valid program in a domain-specific language (DSL). The system uses a grammar-constrained small language model (GC-SLM) combined with grammar-guided decoding to ensure that every generated program conforms to a predefined context-free grammar.

The DSL program is compiled into a compact bytecode representation. A micro-interpreter executes the bytecode on resource-limited embedded devices, such as ESP32-, RP2040-, or ATmega-class microcontrollers. The interpreter is designed with bounded memory usage, deterministic timing, and predictable worst-case execution time.

The invention further includes a fine-tuning technique, Sparse Low-Rank Adaptation (S-LoRA), which introduces structured sparsity into low-rank updates of transformer weights. This allows efficient adaptation of small language models to embedded programming semantics with reduced memory requirements.

The system offers multiple embodiments, including edge/cloud generation with MCU execution, on-device generation using quantized models, multi-grammar support, and FPGA-based interpreters.

---

## BRIEF DESCRIPTION OF THE DRAWINGS

**FIG. 1.** System-level block diagram showing natural-language input, GC-SLM, grammar-guided decoder, DSL output, bytecode compiler, and MCU execution.

**FIG. 2.** Grammar-guided decoding flow showing parser state, token masking, and autoregressive generation loop.

**FIG. 3.** S-LoRA structure showing sparse low-rank matrices and block pruning.

**FIG. 4.** Bytecode interpreter architecture with evaluation stack, instruction dispatch, and hardware pin interactions.

**FIG. 5.** MCU deployment environment including sensors, actuators, communication interfaces, and program update path.

---

## DETAILED DESCRIPTION OF THE INVENTION

### 1. System Architecture

The system comprises:

1. A natural-language front end to collect user intent.
2. A grammar-constrained small language model (GC-SLM) with 1–50 million parameters.
3. A grammar-guided decoding engine enforcing allowed DSL productions.
4. A domain-specific language with a well-defined context-free grammar.
5. A compiler translating DSL constructs into a compact bytecode.
6. A micro-interpreter on the embedded device capable of executing said bytecode deterministically.
7. Optional verification tools to ensure timing, memory, and resource safety.

These components operate together to enable natural-language programming of resource-limited hardware.

---

### 2. Grammar-Constrained Small Language Model (GC-SLM)

A transformer-based language model generates DSL tokens. Unlike standard autoregressive models, the GC-SLM does not freely sample from the vocabulary. At each timestep, the decoder consults a parser state describing the set of valid grammar continuations and masks out invalid tokens. This ensures that the generated sequence is always a syntactically valid program.

The model size is deliberately kept small (1–50M parameters) to enable training on consumer hardware and deployment in edge devices. Training may include natural language, code corpora, and synthetic DSL programs generated automatically from the grammar.

---

### 3. Grammar-Guided Decoding

This component enforces syntactic correctness at generation time. It operates as follows:

1. Maintain a parser state representing the partial derivation.
2. For each generation step, compute the valid next tokens from the grammar.
3. Mask the model's logits to permit only valid tokens.
4. Update the parser state after sampling a token.
5. Terminate only when a complete valid program is formed.

This mechanism eliminates syntactic errors, unbalanced constructs, and malformed programs without requiring post-generation correction.

---

### 4. Sparse Low-Rank Adaptation (S-LoRA)

S-LoRA adapts the base model to embedded programming semantics using low-rank parameter updates with structured sparsity. The low-rank matrices are divided into blocks, and entire blocks may be pruned based on magnitude or learned masks.

Benefits include:

- Reduced memory footprint for adapters.
- Efficient training requiring fewer samples.
- Compatibility with quantization (INT8 or INT4).

- Ability to deploy on devices with minimal on-device storage for fine-tuned components.

---

**5. Domain-Specific Language (DSL)**

The DSL includes constructs for:

- Assignments,
- Event-driven conditionals (“when”),
- Time-based periodic actions (“every”),
- Pin or actuator control,
- Sensor reading,
- Wait or delay operations,
- Local variable manipulations.

The grammar is context-free, enabling deterministic parsing and generation. DSL statements compile to bytecode using a fixed expansion factor, typically between 3 and 8 instructions per statement.

---

**6. DSL Compilation to Bytecode**

A compiler maps DSL constructs to a compact instruction set including operations such as:

- PUSH, POP
- LOAD, STORE
- READ_SENSOR
- SET_PIN
- JUMP, JUMP_IF_ZERO
- WAIT
- HALT

The compiler ensures that the bytecode has predictable size and uses only allowable operations corresponding to the embedded instruction set supported by the interpreter.

---

### 7. Micro-Interpreter

The interpreter executes bytecode with the following characteristics:

- Evaluation stack with fixed maximum depth.
- Instruction dispatch loop with cycle-bound execution.
- Memory footprint under 16 KB.
- No dynamic memory allocation.
- Deterministic worst-case execution time (WCET) per instruction.
- Optional static verification for timing constraints.

The interpreter may run on microcontrollers such as the ESP32, RP2040, ATmega328, STM32, or similar.

---

### 8. Hardware Platform

The embedded device has:

- CPU clock typically in the 50–300 MHz range,
- RAM between 2 KB and 512 KB,
- Flash storage between 32 KB and 4 MB,
- GPIO pins, digital outputs, ADCs, I2C/SPI/UART busses,
- Optional WiFi, BLE, or serial connectivity for program upload.

The system supports both over-the-air updates and wired transfers.

---

### 9. End-to-End Execution Flow

1. User provides natural-language description.
2. GC-SLM + grammar-guided decoding produces a DSL program.
3. DSL program is compiled to bytecode.
4. Bytecode is transferred to MCU.
5. MCU interpreter executes the program deterministically.
6. Sensors, actuators, and peripherals respond to program logic.
7. Execution may loop indefinitely or until HALT.

---

**10. Safety and Validation Embodiments**

The invention may include mechanisms such as:

- Pin sanity checks (e.g., preventing writing to restricted pins).
- Rate limits on PWM or output frequency.
- Bounds checking of sensor values.
- Static analysis of DSL programs to determine schedulability.
- Enforcement of maximum instruction counts in loops.

These mechanisms ensure program reliability in safety-critical contexts.

---

**11. Alternative Embodiments**

This invention encompasses variations including:

**A. On-device generation** A quantized 4–10M parameter model stored in MCU Flash generates DSL locally.

**B. FPGA interpreter** Interpreter realized as hardware logic for higher determinism.

**C. Multi-grammar support** The system loads multiple grammars at runtime for different device domains.

**D. Hybrid verification** Formal methods check bytecode correctness before deployment.

**E. Richer DSL constructs** State machines, streaming pipelines, or event-driven graphs.

These embodiments extend the scope of the invention without departing from its principles.

---

## ADVANTAGES OF THE INVENTION

- Enables natural-language programming of embedded devices.
- Guarantees syntactic correctness of all generated programs.
- Provides deterministic, low-memory execution suitable for microcontrollers.

- Reduces training compute and storage via S-LoRA.
- Allows rapid prototyping and field updates without recompiling C/C++.
- Operates offline, without requiring a cloud-based large language model.
- Supports safety-critical and real-time embedded applications.

---

## EXAMPLES

### Example 1: LED Blinking on Condition

NL Input: “Blink the LED on pin 2 every 500 milliseconds only when the button on pin 12 is pressed.”

Generated DSL (example):

```
when read pin 12 == 1:
   every 500ms:
       set pin 2 to 1
       wait 200ms
       set pin 2 to 0
```

Compiled to bytecode and executed on MCU with deterministic timing.

### Example 2: Temperature-Based Fan Control

NL Input: “When temperature exceeds 30, run the fan at half power.”

DSL:

```
when sensor.temp > 30:
   set fan to 128
```

---

## CONCLUSION

The invention provides an integrated system combining grammar-constrained small language models, deterministic compilation, and a resource-bounded interpreter to enable reliable, natural-language-driven programming of embedded devices.