FROM WORDS TO WIRES: Generating Functioning Electronic Devices from Natural Language Descriptions

Peter Jansen
University of Arizona
pajansen@arizona.edu

Abstract

In this work, we show that contemporary language models have a previously unknown skill – the capacity for electronic circuit design from high-level textual descriptions, akin to code generation. We introduce two benchmarks: PINS100, assessing model knowledge of electrical components, and MICRO25, evaluating a model's capability to design common microcontroller circuits and code in the ARDUINO ecosystem that involve input, output, sensors, motors, protocols, and logic – with models such as GPT-4 and Claude-V1 achieving between 60% to 96% PASS@1 on generating full devices. We include six case studies of using language models as a design assistant for moderately complex devices, such as a *radiation-powered random number generator*, an *emoji keyboard*, a *visible spectrometer*, and several *assistive devices*, while offering a qualitative analysis performance, outlining evaluation challenges, and suggesting areas of development to improve complex circuit design and practical utility. With this work, we aim to spur research at the juncture of natural language processing and electronic design.[1][2]

1 Introduction

The realm of science fiction often presents us with captivating visions of technology's future. A case in point is the *replicator* from the TV series *Star Trek*, a machine capable of creating various physical objects – from food and medicine to functioning devices – based solely on a user's high-level description of those objects. Contemporary language models hint at the precursors to some of this capacity for design, including the ability to design novel 2D and 3D object models (Ramesh et al., 2022; Nichol et al., 2022), predict molecular structures for drug discovery (Liu et al., 2021; Flam-Shepherd and Aspuru-Guzik, 2023), and generate increasingly complex sections of code that power a variety of user applications (Li et al., 2023; Wang et al., 2023).

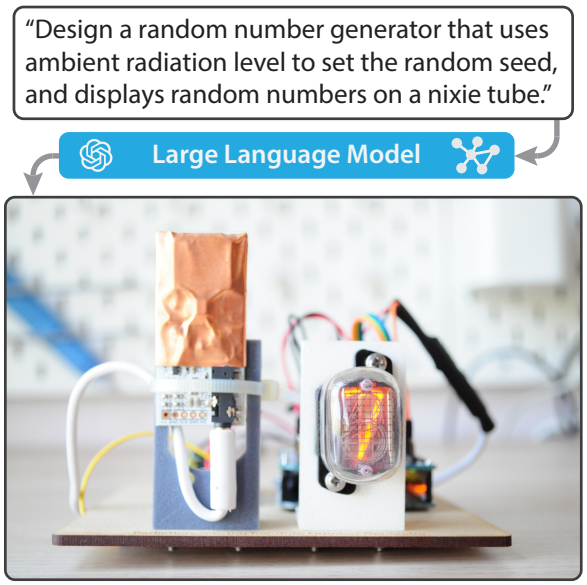


Figure 1: An example of using a language model to convert a high-level textual description of an electronic device into the designs for that device. Those designs are then reviewed by a domain expert, and any errors corrected, before the device is physically manufactured using rapid-prototyping techniques.

In this work, we show that language models have a previously unknown skill – the capacity to generate working electronic devices from high-level textual descriptions – effectively bridging the gap between the *words* of a device description to the *wires* of a device design. The design process for electronic devices, such as the random number generator in Figure 1, typically follows a stage-like process illustrated in Figure 2. These steps include: ideation, electronic design (including generating parts lists, electronic schematics, and code for embedded processors called *microcontrollers*), followed ultimately by physical implementation

[1] WORDS2WIRES library: https://github.com/cognitiveailab/words2wires
[2] Companion video: https://youtu.be/PZ1rr0dDAPI

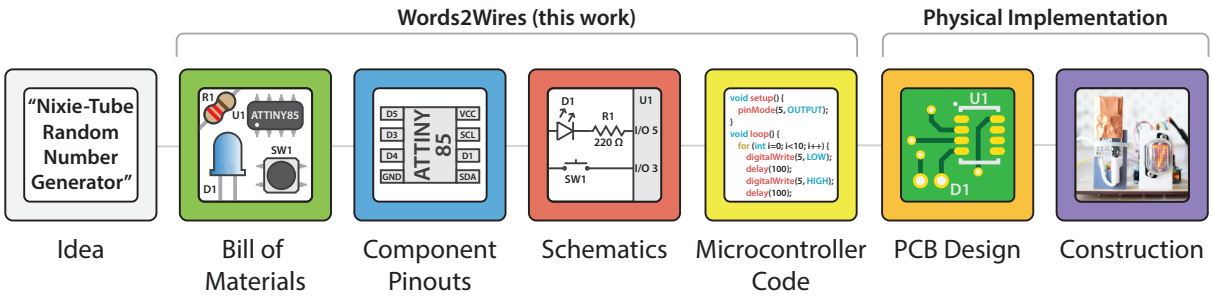


Figure 2: An overview of the electronics device design process, from concept to design implementation.

via manufacturing. Our focus in this work is on automating the task of transforming a high-level concept into a practical electronics schematic, complete with companion microcontroller code—a task that currently requires significant human expertise and effort. We evaluate our approach by constructing these devices either in simulators or, for six open-ended case studies, as physical devices implemented using rapid prototyping techniques such as breadboards, circuit board manufacture, laser cutting, and 3D printing.

The contributions in this work are:

1. We empirically demonstrate the novel capacity for language models to design electronic devices from high-level textual descriptions, and introduce two benchmarks to measure this capacity. The first, PINS100, measures knowledge of 100 common electronic components. The second, MICRO25, measures the ability to design 25 common electronic devices from start to finish, including the generation of *bills-of-materials*, *pinouts*, *schematics*, and *code*.

2. Our experimental evaluation shows that both GPT-4 and CLAUDE-V1 have moderate-to-strong performance on these benchmarks, with GPT-4 achieving 96% PASS@1 at generating correct schematics and functioning code on MICRO25, while CLAUDE-V1 scores 60% on schematics and 76% on code.

3. We present six real-world case studies of using language models to construct novel devices, including an *emoji keyboard*, a *visible light spectrometer*, and two *assistive devices*. Alongside, we provide a qualitative assessment of the strengths of current language models for electronic device design, while also outlining the challenges in enhancing performance, automating evaluation, and increasing their practical utility.

2 Related Work

Arduino microcontroller ecosystem: This work focuses on circuits that are controlled by *microcontrollers*, which are processors intended for embedded applications. In particular, we focus on microcontrollers supported by the ARDUINO ecosystem (Banzi and Shiloh, 2022), a popular cross-platform set of C++ tools and libraries[3] intended to promote learning and lower the barrier to entry for creating projects. Most of the devices described in this work target the ARDUINO UNO, which uses an ATMEGA328P microcontroller with 2K of RAM, 32K of program space, 20 input/output (I/O) pins, and a speed of 16MHz.

Code Generation: Though many aspects of the electronic design process have been semi-automated using mature technologies for decades, particularly as they relate to physical design (e.g. integrated circuit and circuit board layout; see Huang et al. (2021) for review), to the best of our knowledge this is the first work that explores using language models to automate circuit design at the early design stage – from a textual description of a device, to an initial draft of an electronic schematic and companion microcontroller code.

Designing code-driven electronic circuits is similar to code generation tasks (e.g. Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Lai et al., 2022; Nijkamp et al., 2023, inter alia), with the additional requirement that a model must jointly generate a corresponding circuit that is compatible with the code, that together allow the microcontroller to accomplish a given task. Code generation models powered by large language models have progressed rapidly in the last several years, particularly in terms of generation length, with systems progressing from generating a few lines of code (Chen et al., 2021; Li et al., 2022; Fried et al., 2023)

[3] https://www.arduino.cc/

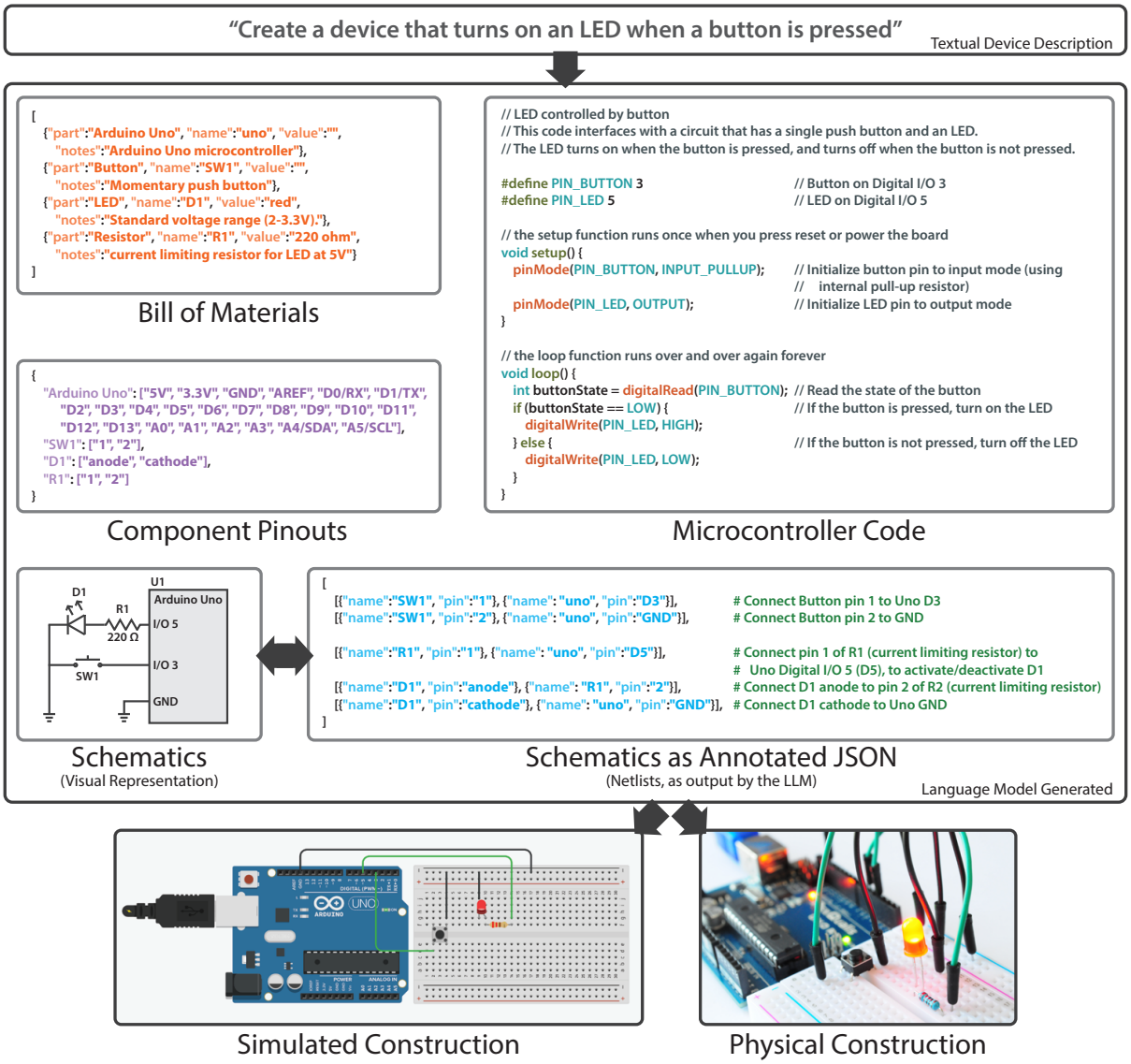


Figure 3: An example of representing a device specification as text (in formatted JSON) such that it can be generated by a language model, for a trivial device that illuminates an LED in response to a button being pressed. The device specification includes the bill of materials, component pinouts, schematic (represented as a netlist), and microcontroller code. The specification can then be used to create that device, either in simulation or through physical construction. The device shown here was generated by GPT-4, and edited slightly for space.

to producing larger segments containing dozens or hundreds of lines of code (OpenAI, 2023; Li et al., 2023) within a short period. This increase in performance is due in part to an increase in model context lengths to 8K tokens, which also enables this work – as generating bills of materials, pinouts, and schematics in addition to code requires a large number of additional tokens.

In this work, we investigate device generation in two contexts. First, we assess single-turn generation of error-free devices using the MICRO25 benchmark, a set of electronic design tasks evaluated using code metrics similar to PASS@1 (Kulal et al., 2019) that require the model to generate a single correct solution, and use strict binary measures of task success. In the second context, we explore a more collaborative coding-assistant setting, where prompts can be iteratively refined, and any errors in the schematics or code corrected by the end user. This approach is akin to tools such as GITHUB COPILOT (Chen et al., 2021), which assist in generating short parts of programs. However, in our case, the model is utilized to generate an initial version of the entire project, after which the user corrects any errors before physically constructing the project.

Pinout Scoring Method	GPT3.5	GPT4	Claude
Strict Scoring (Exact)	55%	74%	56%
Permissive Scoring	74%	86%	72%

Table 1: Model accuracy in generating accurate pinout information for 100 common electronic components. *Strict scoring* requires all generated pins on a given device to be accurate in order to be counted as correct. *Permissive scoring* requires pins critical to the function of a given component to be correct, but still counts generations with non-critical missing pins (such as when the device is mounted to a breakout board) to be correct.

3 Experiment 1: Component Knowledge

To design electronic circuits, the designer needs a knowledge of the individual electrical components that can be used to build a circuit. One of the most fundamental aspects of this knowledge is the *component pinouts*, or the specific function of each electrical terminal (or *pin*) on a component. For example, a light-emitting diode (LED) typically has two pins, one an *anode* where positive voltage is applied, and one a *cathode* where the negative terminal or ground is applied. Here we measure large language models' knowledge of component pinouts by asking them to generate pinouts for a large number of common electrical components, and verifying their accuracy.

Component Pinouts: While LEDs and other common components such as resistors and capacitors commonly have two pins, other components have varying numbers of pins, each with different functions. For example, a common kind of digital motor called a *hobby servo* typically has three pins: an anode, a cathode, and a digital signal pin that accepts a pulse from a microcontroller that signifies what angle the motor should turn to. Common sensors generally have between two and ten pins, depending on their mode of function, and the communication protocols (e.g. digital vs analog) they employ. Similarly, integrated circuits, and microcontrollers in specific, can have as few as 8 pins (such as the ATTINY85 microcontroller in Figure 2), but frequently have dozens and occasionally hundreds of unique pins. Before a circuit can be designed using a given component, the functions of each pin on that component must be known so they can be appropriately connected to other components toward creating a desired function. Connecting components incorrectly will, at best, create a non-functioning device, while at worst will commonly destroy a given component, or the entire device – so this knowledge is critical.

Benchmark: We assembled a benchmark of electronic component pinouts, PINS100, containing 100 common parts frequently used in circuits found on high-traffic electronic tutorial websites such as the ARDUINO PROJECT HUB and AUTODESK TINKERCAD CIRCUITS. Components range from 2 pins to 40 pins, and span a large assortment of part categories including passives (e.g. *resistors/capacitors*), input (e.g. *switches*), output (e.g. *LEDs, motors, relays*), sensors, integrated circuits, power regulators, logic (e.g. 7400-SERIES *AND and OR gates*), and microcontrollers (e.g ARDUINO, RASPBERRY PI).

Models: We evaluate on instruction-tuned models including OpenAI's CHATGPT (GPT-3.5-TURBO) and GPT-4 (OpenAI, 2023), and Anthropic CLAUDE-V1 [4]. Model prompts are identical across models, and include a static 1-shot exemplar (a 14-pin 7400-SERIES logic integrated circuit) that provides an example of the pinout task, as well as the requested JSON output format. Additional model details and hyperparameters are provided in the APPENDIX.

Evaluation: We evaluate using two binary measures of accuracy analogous to PASS@1 (Kulal et al., 2019). The first scoring method, *strict*, requires a given model to output all of a component's pins correctly to be considered correct, otherwise it will be considered incorrect. The second method, *permissive*, requires only the function-critical pins of a component to be present to be considered correct, while failing to include rarely-used or non-critical pins still counts as success.

Two additional considerations complicate evaluation. First, the same electrical component may come in different *packages*, or be built by different manufacturers – for example, the power pin (e.g. VCC) on a given component might be on pin 3 in one package, and pin 5 on a different package. As such, evaluation requires only that the pin name be correct (e.g. "VCC"), but does not require producing pin numbers, which are typically matched when choosing a particular component package during the circuit board design phase. Second, the specific names for pins are often described differently – and frequently with only single letters. For

[4] https://www.anthropic.com/

example, a pin with a *reset* functionality might be described variously as "RESET", "RST", or even simply "R" in different sources of text, such as official part datasheets or web tutorials, with each label being correct. As such, model-generated output is evaluated fully manually by a domain expert, who requires either generated pin names to match official documentation, obvious short forms, or (in the case of large differences) alternative part naming conventions found through a web search. This alignment process mirrors the actual electronics design process, where a reference schematic for a circuit may use different pin names than an official datasheet, and these pins need to be manually aligned by a domain expert by searching through a variety of reference materials.

Results: Model performance in the pinout generation task is shown Table 1. Performance reflects average binary PASS@1 performance of a given model on generating accurate pinouts – for example, a score of 50% reflects that 50% of the components had completely correct pinouts. Here, GPT4 achieves the highest *strict* scoring performance, generating accurate pinouts for 74% of components, while GPT3.5 and CLAUDE-V1 achieve similar levels of performance, generating correct pinouts for 55% and 56% of components, respectively. *Permissive* scoring increases performance, with the best-scoring GPT-4 model capable of generating pinouts that include the most critical pins for 86% of electrical components in the benchmark. Taken together, these results suggest that large language models have a moderate knowledge of electrical component pinouts, a core requirement for designing functioning electronic circuits.

4 Experiment 2: Circuit Generation

While we've observed that contemporary language models have a moderate knowledge of electrical component pinouts, how well can they leverage that knowledge to design functioning simple electronic devices? In this experiment, we investigate end-to-end generation of working devices, which includes generating four core components: (1) a *bill of materials*, or list of components in the device, (2) the *pinouts* for each component, (3) a complete electrical circuit diagram called a *schematic*, that details how the components are to be connected, and (4) the *code* to be programmed onto a microcontroller – a light-weight processor that controls embedded circuits. Examples of a full generated design for a trivial device that turns on an LED in response to a button being pressed is shown in Figure 3.

Benchmark: To assess a model's ability to create microcontroller-driven electronic devices, we developed a benchmark, MICRO25, that includes 25 tasks intended for the common ARDUINO microcontroller ecosystem.. These tasks, shown in Table 2, span 5 core categories including: input, interface protocols, output, sensors, and logic. Each task is either tailored to test a specific fundamental competency required to build basic microcontroller-driven electronic devices, or the integration of several competencies into larger design flows.

Representations: Models were given format prompts to export all generated components (*bill of materials*, *pinouts*, *schematics*, and *code*) in an annotated JSON format, shown in Figure 3. The annotated format allows the model to add comments for each generated element (e.g. specifying the uses of each component, or the purpose of each connection in the schematic), analogous to chain-of-thought reasoning (Wei et al., 2022) applied to circuit generation.

The *bill of materials* format expresses canonical information typical in the design process, including the component type (e.g. "resistor"), component name in the schematic (e.g. "R1"), component value (e.g. "10k ohms"), as well as a note on the purpose of the component (e.g. "current limiting resistor for LED"). The *pinouts* are expressed as a dictionary containing lists of pins for each part, as in Experiment 1. *Code* is expressed between MARKDOWN code blocks to ease extraction. *Schematics* are expressed as "netlists", which are a common storage format frequently adapted by electronic design tools. This format is analogous to an undirected graph, where edges represent a given connection from one component pin (such as the anode of an LED) to another component pin (such as one terminal of a current-limiting resistor).

Models: We evaluate on instruction-tuned models with large (8k token) context windows, including OpenAI's GPT-4 (OpenAI, 2023) and Anthropic CLAUDE-V1. Model prompts are identical across models, and include a static minimal 1-shot example of generating each of the 4 components (*bill of materials*, *pinouts*, *schematic*, *code*) in the desired JSON output format. In response to specific types of errors identified during pilot studies, the prompt also includes three incomplete snippets that

Category	Task Description	GPT4 - Schematic	GPT4 - Code	Claude - Schematic	Claude - Code
Input					
Digital - Button	Turn on an I/O pin when a single button is pressed.	✓	✓	✓	✗
Digital - Multiple	Turn on an I/O pin when exactly 2 of 4 buttons are pressed.	✓	✓	✓	✗
Analog	Read potentiometer, activate I/O pin when voltage exceeds 2.5V	✓	✓	✗	✓
Protocols					
Serial/UART	Read the serial port. When "hello" is read, respond with "world".	✓	✓	✓	✓
Serial/SPI	Connect using SPI. When "hello" is read, respond with "world".	✓	✓	✓	✓
I2C	Read a value from a specific I2C address and register every second.	✓	✓	✓	✓
SD Card	Open a file, randomly append one of 4 strings every 10 seconds.	✓	✓	✓	✓
Output					
Motor - DC	Rotate a DC motor clockwise then counterclockwise every 5 seconds.	✗	✓	✗	✓
Motor - Stepper	Oscillate a stepper motor 45 degrees every 5 seconds.	✓	✓	✗	✗
Motor - Servo	Continuously move a servo back and forth from 45 to 135 degrees.	✓	✓	✓	✓
LED - Blink	Blink an LED every 500 milliseconds.	✓	✓	✓	✓
LED - Sequence	Blink 4 LEDs in sequence, once every 500 milliseconds.	✓	✓	✗	✓
LED - 7 Segment	Count from 0 to 9 on a 7-segment display, changing every 500 msec.	✓	✓	✓	✓
LED - Neopixel	Slowly move through a rainbow of colours on an RGB LED.	✓	✓	✓	✓
Relay	Turn on a relay for 2 seconds, then off for 5 seconds.	✓	✓	✗	✓
LCD	Print "Hello World" on a 16x2 LCD (HF44780 compatible controller)	✓	✓	✗	✓
Sound - Buzzer	Cycle between high, medium, then low tones on a Piezo Buzzer.	✓	✓	✗	✓
Analog Output	Produce a sawtooth wave, ramping up from 0V to 5V every 50 msec.	✓	✓	✗	✓
Sensors					
Resistive - CDS	Read the light intensity from a resistive sensor (CDS cell).	✓	✓	✗	✓
Manual Protocol	Read the distance from a HC-SR04 ultrasonic distance sensor.	✓	✓	✓	✓
I2C - Magnetic	Read x/y/z components of magnetic field using an I2C magnetometer.	✓	✗	✓	✓
Logic					
Simon	Create the popular memory game Simon, using 4 colors.	✓	✓	✓	✗
Conway	Create the popular Conway's Game of Life, on an 8x8 LED matrix.	✓	✓	✗	✗
Clock	Create a clock on a 16x2 I2C LCD, with buttons for setting the time.	✓	✓	✓	✗
Air Temperature	Read the air temperature, display temperature on an RGB LED.	✓	✓	✓	✓
Overall Performance		96%	96%	60%	76%

Table 2: Model performance (PASS@1) on the MICRO25 benchmark generation tasks, broken down by *schematic* and *code*. Task descriptions are summarized for space, where full task descriptions can be found in the benchmark.

provide portions of two positive and one negative generation example.[5] After initial generation, the models are given a reflection prompt containing 12 common errors (such as correctly supplying power to each component, explicitly enumerating each connection in the schematic, and having code that functions as intended), and allowed to iteratively reflect and improve output until providing a specific stop token signifying that the model has detected no further errors. The initial prompt requires 1884 tokens, and reflection prompt requires 431 tokens. Additional model details are provided in the APPENDIX.

Evaluation: Generated devices are broken down into electrical (schematic) and code components, each of which is separately evaluated using a binary PASS@1 metric (i.e. functioning/non-functioning). Automatic evaluation faces a number of challenges in this domain, including that many different solutions are possible for each task, and existing simulators typically lack many of the possible components a model might generate in a solution. As such,

[5] Snippets rather than full examples are used to decrease the number of tokens used in the prompt. The static example in the 1-shot prompt is a device that blinks two LEDs in an alternating pattern. The snippets include: connecting a servo motor (a device with 3 pins), connecting a button with a pull-up resistor (a situation where 3 pins – the microcontroller input, a resistor terminal, and a button terminal – are interconnected), and a negative example where the schematic takes a shortcut and connects ranges of pins in a single line instead of enumerating each connection individually.

evaluation was conducted manually by a domain expert. The schematics and code were manually inspected for functionality. When non-trivial or uncommon solutions were generated, the circuits were evaluated by constructing them in a simulator (AUTODESK TINKERCAD, shown in Figure 3) when possible, or physically building the circuits when not possible. When circuits used difficult to source or obsolete components, evaluation occurred through manual inspection, and comparing the generated schematics and code to reference materials. For a schematic to be considered correct, it must contain all relevant components, and be wired correctly in a way where code could be written to accomplish the desired task. For code to be considered correct, it must correctly perform the task given the generated schematic – or for cases where the schematic was incorrect, be able to accomplish the task as-is were the schematic corrected.

Results: The results of the full circuit generation task are shown in Table 2. GPT-4 performs extremely well on this benchmark, correctly generating schematics and code for 96% of benchmark tasks. CLAUDE-V1 demonstrates more modest performance, achieving 60% for schematic generation, and 76% for microcontroller code generation. Taken together, this shows that contemporary language models have moderate-to-excellent overall capacity for generating common electrical circuits end-to-end, from bills of materials, pinouts, and schematics, to paired microcontroller code that accomplishes the desired functionality.

5 Experiment 3: Open Device Generation

While we've observed that language models have the capacity to design comparatively simple devices in Experiment 2, this result is tempered by these benchmark tasks being representative of fairly common skills and capacities that are frequently taught in microcontroller-oriented curricula found in books, internet tutorials, and blog posts – and as such, the MICRO25 tasks likely exist in some form in the voluminous (but closed) training data of these models. Here, we examine how well the best-performing model, GPT-4, can create comparatively more complex devices in a more realistic and qualitative setting, where it is used as a *design assistant* like GITHUB COPILOT (Chen et al., 2021) to create initial plans that are then vetted and corrected by a domain expert before being physically constructed. To further increase task difficulty, each of the device specifications in this experiment were explicitly crafted to be highly unusual – either using uncommon components, or combining common components in unusual ways – such that the likelihood of similar devices appearing in the closed model training data is low.

Benchmark: Six devices, crafted to use uncommon components, or common components in uncommon ways. The six devices are:

1. **Random number generator:** a random number generator, using two uncommon components: (a) a radiation sensor to help provide a random seed based on ambient radiation levels, and (b) a high-voltage nixie tube (or cold-cathode display), similar to a vacuum tube, to display randomly generated digits. Nixie tubes were manufactured and used in the 1950s and 1960s before light-emitting diodes became common.

2. **Emoji keyboard:** a USB keyboard that only contains keys for common emoji characters.

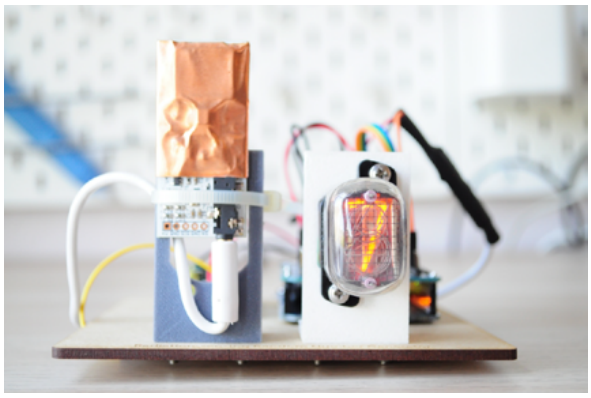
3. **Spectrometer:** a visible light spectrometer using the uncommon Hamamatsu microspectrometer, and that displays the spectrum on an organic LED (OLED) display.

4. **Non-contact temperature:** a device that measures the temperature of an object using a common infrared-based non-contact temperature sensor, but displays the temperature in an uncommon way: as a color-changing bar graph on an LED display.

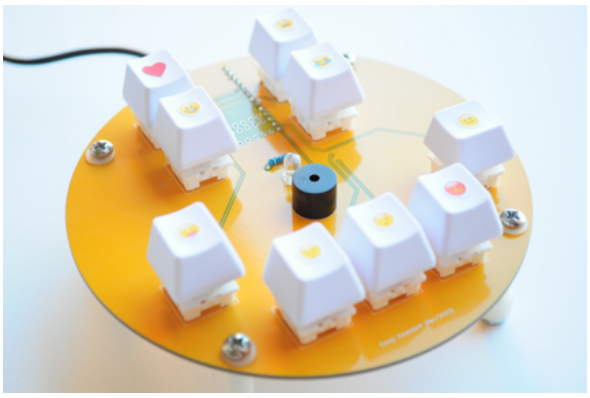
In addition, two assistive devices were explored:

5. **Pill alarm:** a common pill-alarm, that displays the current time on an LCD display. The alarm is presented in an uncommon way: by using servo motors to physically waive flags that say *"take pill X"* for hearing-impaired users, until they press a reset button.

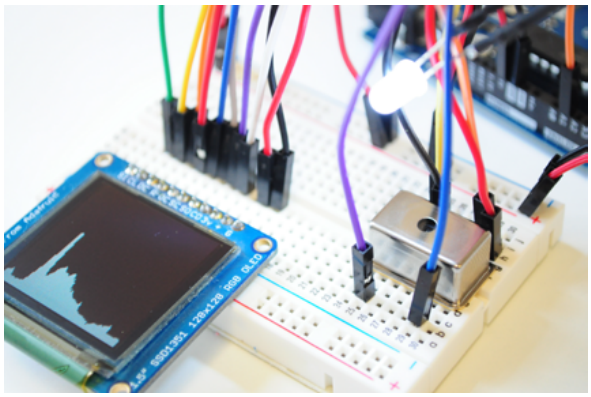
6. **Ultrasonic glasses:** common components (an ultrasonic distance sensor and piezo buzzer) used for an uncommon purpose – to create a pair of glasses for the visually-impaired. The glasses audibly notify the user of the distance to objects in front of them using a tone whose frequency varies with distance.



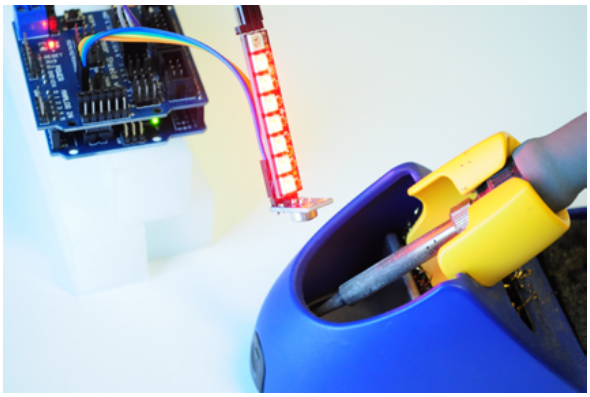
Random number generator that uses an ambient radiation sensor to continuously update the random seed. Random numbers are generated every few seconds, and displayed on a vintage nixie tube using a high-voltage driver.



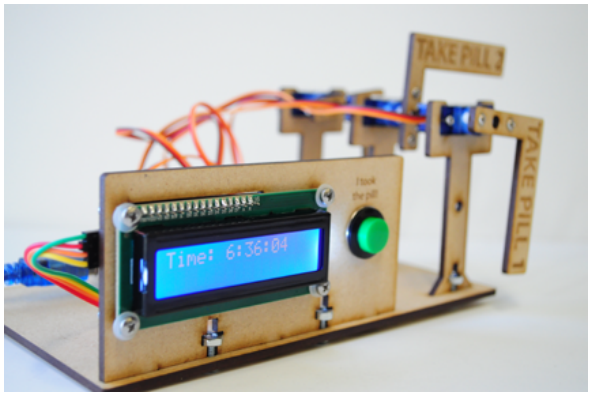
Emoji USB keyboard that has keys for 9 common emojis. Pressing an emoji types its ASCII string, just as if entered on a normal keyboard. A short musical tune with similiar affect to the emoji (e.g. a love song for the heart emoji) is also played.



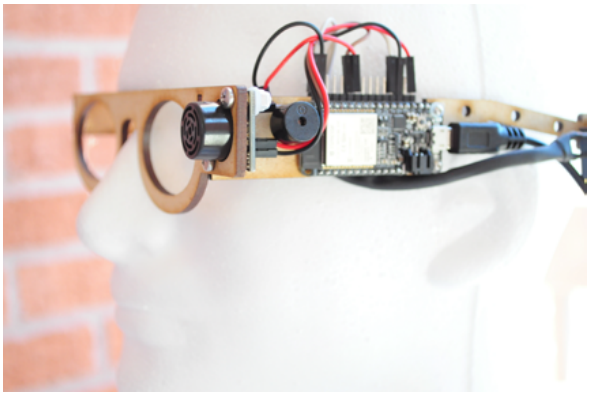
Visible light spectrometer that measures spectra using a Hamamatsu micro-spectrometer and displays the spectrum on a 128x128 pixel OLED screen. Here, the device is shown measuring the characteristic spectrum of a white LED.



Non-contact temperature sensor that displays the temperature on a strip of 8 LEDs. Higher temperatures show as red colors and illuminate more LEDs, while lower temperatures show as blue colors and illuminate fewer LEDs.



Pill alarm assistive device that has alarms for three pills. When it's time to take a pill, the alarm continuously waives a physical flag saying "Take Pill *X*" back and forth using a servo motor to get the users attention, until the button is pressed.



Ultrasonic glasses assistive device that uses an ultrasonic distance sensor to measure the distance of the nearest object in front of a person with a visual impairment. The distance is converted into an audible tone of varying frequency.

Figure 4: Six devices designed using WORDS2WIRES in the open generation condition, then physically constructed.

Methods: Initial specifications (in the form of a natural language textual description of a device) were created for all devices, and iteratively refined several times to provide clarifications in response to undesired or errorful model-generated device specification output. After several attempts at refining textual device descriptions, any remaining errors were manually corrected by a domain expert, then the devices were physically manufactured. All devices were constructed using prototyping techniques such as solderless breadboards and jumper wires, with the exception of the *emoji keyboard*,

which was designed as a printed circuit board using the KiCad electronic design automation (EDA) suite, then sent for manufacture. Mounting enclosures (centrally for the *random number generator*, *pill alarm*, and *ultrasonic glasses*) were human-designed, and manufactured using rapid-prototyping technologies (laser cutting and 3D printing). The final device builds are shown in Figure 4, with qualitative observations of design process challenges identified below.

5.1 Qualitative Challenges

Overall high-level qualitative challenges in designing the six case study devices are described briefly below, where a detailed description of errors and corrections for each of the six devices is provided in the Appendix.

Sensitivity to prompt: Small and seemingly helpful changes in the device description or prompt can cause large changes in generation. For example, for the *random number generator*, including a reminder that the radiation sensor required a pull-up resistor appeared to cause the model to forget to include a high-voltage supply for the nixie tube.

Many requirements can create poor performance: Adding many composite requirements to a project, even when they are individually easy, can create low performance. For example, for the *emoji keyboard*, adding the requirement to play a relevant musical tune when each key is pressed generally produced only scaffolds for music generation code without actually including the melodies. A subsequent call to GPT-4 asking it only to fill in this music scaffold was required to generate the melodies.

Device drivers: The model performs best for straight-forward circuits where the coding portion of interfacing with external components (such as sensors) is abstracted to existing libraries. When writing a low-level device library is required, the model commonly either hallucinates a non-existent library, or generates a reasonable first-pass at a device library that requires extensive modification to function. For example, the *microspectrometer* device driver the model generated had the essential conceptual-level components – i.e. that data needed to be clocked out of the spectrometer and read by an analog-to-digital converter after sending a start pulse to the spectrometer – but the generated code had incorrect clock timings, logic levels, and other fine-grained details which made it unable to function without correction.

Deprecated or mismatched libraries: Hardware libraries are frequently updated to support new features, but the Arduino ecosystem lacks *Makefiles* or a build manager with explicit library version numbering, making specific library versions unknown for a given code example. During manufacture, this frequently required searching through old versions of a library (such as the *Bounce* button input library for the *emoji keyboard*) to find a version that matched the specific API generated by the model. Similarly, observing examples of different APIs across different library versions causes the model to occasionally mix APIs from older and newer versions of libraries, or from different libraries with similar functions (such as combining the APIs of several LCD display libraries for the *spectrometer*).

End-of-life parts: Electronic components regularly reach end-of-life cycles, and are no longer manufactured or easily available. The model occasionally generated circuits that used unavailable parts, and had much less competency generating circuits for newer part variants, particularly those that were released near GPT-4's knowledge cutoff date of September 2021 (OpenAI, 2023).

6 Discussion

In this work, we show that contemporary language models have a previously unknown skill – the capacity for electronic circuit design. Within that context, we identify several opportunities and challenges for further development:

Devices generated with common-sense knowledge: Large language models contain common-sense knowledge, and knowledge outside the domain of electronics. For example, the model could generate a keyboard for common emojis without being told what an *emoji* is, the relative frequency of different *emojis*, or their sentiment (for the melody-generation subtask). Similarly, the model could extrapolate the function of creating an assistive device for the blind, simply by being told that it required generating a "helpful sound" that corresponds to how close something is in front of the person.[6] These and similar applications that

[6] Albeit, with a model-generated sound that is likely more palatable (and useful) to robots than humans, for the time being.

seamlessly blend device design with human-level common-sense or domain knowledge *outside of electronics* may ultimately substantially lower the barrier to creating customized devices for arbitrary applications – where this is currently uncommon, in large part due to the expense of the associated design work.

Speeding electronic design: Just as coding assistants such as GITHUB COPILOT can increase human productivity for coding tasks (Ziegler et al., 2022), the use of a suite of electronic design assistants may similarly increase productivity in electronic device design, reducing the design process from days to hours (or, minutes). But, for complex devices, this utility may be tempered in the near-term by the time required to review every aspect of the design for accuracy.

Automating physical design: This work focuses on only part of the design process – generating bills-of-material, pinouts, schematics, and code. A significant aspect of design includes *physical design*, such as the design of a printed circuit board to hold all the device components, or an enclosure and any mechanical components of the device. Parts of this process, such as automatically routing circuit boards, are mature technologies that have existed for decades (e.g. Huang et al., 2021). Others, such as enclosure design, might leverage the rapidly expanding capabilities of visual and spatial models (e.g. Nichol et al., 2022) to make rapid progress in this area – though this process may be tempered, at least in part, by the necessity of having detailed physical dimensions of all components used in a device. Taken together, many of these barriers appear to be solvable technical problems that might be mitigated by the creation of open-source part databases that include physical dimensions, and are areas of immediate action for future work.

Simulating designs: Like COPILOT, we have observed that the electronic designs generated by language models are rarely perfect and frequently have errors. Currently these have to be discovered and corrected by a human. Contemporary work in code generation aims to use reflection to iteratively run generated code in an external interpreter (like PYTHON), report any errors to the language model, then continue this process until the generated code runs error-free. There are significant barriers to making this possible for electronic device generation tasks, chief among them the lack of electronic simulators with large libraries of simulated devices for any part a language model is likely to generate, such as sensors, integrated circuits, or other components. This is a significant barrier to automatically *evaluating* and refining circuits, and could begin to be addressed by creating an open-source simulator with a standard definition of component models, that manufacturers could choose to release component models for. Similarly, a hybrid approach could be effective, where the model is prompted to only use parts that are included in a given simulator's repertoire.

7 Conclusion

In this work we empirically characterize the previously unknown capacity for contemporary language models to move from *words* to *wires* – that is, to generate working electronic device designs from high-level device descriptions. We demonstrate that models such as CHATGPT, GPT-4, and CLAUDE-V1 have a moderate knowledge of electronic component pinouts on the PINS100 benchmark, while GPT-4 substantially outperforms CLAUDE-V1 at generating full working devices on the MICRO25 tasks, reaching near-perfect performance.

When used as a design assistant for generating more complex devices, language models are capable of generating devices that *nearly* meet specifications, but currently require modest to moderate correction by human domain experts to both electrical (schematic) and code designs before being functional. While using language models to design electronic devices from high-level textual descriptions opens exciting possibilities in de-skilling the creation of physical devices – potentially increasing accessibility, or increasing the speed of domain experts as design assistants – further development is currently tempered by the lack of simulators to automatically evaluate designs, and the highly manual (and time-consuming) nature of this process.

8 Limitations

This work has a number of limitations, including:

Device scope: The devices generated in this work are small in scope, with limited functionality – typically a small number of components, fewer than 50 lines of code, and controlled by ARDUINO microcontrollers which are frequently limited to only 2K of memory. This work does not address designing moderate or complex devices such as phones, personal computers, or other devices that are orders of

magnitude more complex in terms of component counts and code length.

Generation accuracy: While the simple devices in Experiment 2 can reach high generation accuracy, particularly with GPT-4, nearly all devices in the more complex open generation condition in Experiment 3 contained errors, and required correction by a domain expert.

Physical design and manufacture: The physical manufacturing of the devices – including building circuits on prototyping breadboards using jumper wires, designing printed circuit boards, or designing physical 3D printed or laser cut enclosures, was entirely manual and completed by a human. While technologies (such as autorouting) exist to automate some of these aspects, they were not used in this work.

Safety: Constructing electronic devices has real dangers and potential harms, including but not limited to the risk of fire, electrical shock, and equipment damage, and should not be attempted by non-experts. The development environment is notoriously hostile to components, and even experienced electrical engineers frequently face safety challenges or accidentally destroy components. Generated devices should always be vetted by a domain expert, and not used for safety-critical applications, or applications where harmful unintended effects may be possible.

Acknowledgements

Thanks to Peter Clark for helpful comments on a draft of this work. We thank the Allen Institute of Artificial Intelligence (AI2) for funding this work.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Massimo Banzi and Michael Shiloh. 2022. *Getting started with Arduino*. Maker Media, Inc.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Daniel Flam-Shepherd and Alán Aspuru-Guzik. 2023. Language models can generate molecules, materials, and protein binding sites directly in three dimensions as xyz, cif, and pdb files. *arXiv preprint arXiv:2305.05708*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. 2021. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(5):1–46.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *arXiv preprint arXiv:2211.11501*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago,

et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zhichao Liu, Ruth A Roberts, Madhu Lal-Nag, Xi Chen, Ruili Huang, and Weida Tong. 2021. Ai-based language models powering drug discovery and development. *Drug Discovery Today*, 26(11):2593–2607.

Alex Nichol, Heewoo Jun, Prafulla Dhariwal, Pamela Mishkin, and Mark Chen. 2022. Point-e: A system for generating 3d point clouds from complex prompts. *arXiv preprint arXiv:2212.08751*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

OpenAI. 2023. Gpt-4 technical report.

Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29.

A Experiment Hyperparameters

Models (GPT3.5, GPT-4, CLAUDE-V1) used similar hyperparameters through all experiments, with precise configurations and APIs available on the Github repository. Models used greedy decoding *(temperature = 0)* to make experiments near-deterministic, though model output still does change between successive runs. Specific (cached) model output for each experiment is provided in the GITHUB repository.

B Error Analysis: Modifications to Open Generation Devices

The devices generated in the open generation experiment generally required modification to function as intended. Here we provide a list of the major design changes required to reach functionality:

B.0.1 Random number generator

Both the radiation sensor (*Radiation Watch Type 5*) and the nixie tubes (*IN-12A*) are highly uncommon components, and are likely to have limited examples available in existing documentation. Generally, across several iterations of device description prompts, either the radiation sensing circuit was correct, or the nixie tube circuit was correct, but not both. The radiation detector requires a pull-up resistor to function, and is pulled low when a high-energy particle strikes it. The nixie tube requires an external high-voltage driver, which was usually generated correctly, but when generated incorrectly it was typically powered by USB voltage (5V) instead of the required high-voltage (170V). Across device descriptions, the method used in the code to set the random seed based on the radiation sensor varied – some useful, some largely incorrect.

B.0.2 Emoji keyboard

The base emoji keyboard was largely generated without issue, though did mix up the version for the button input library, and failed to mention the special programming requirements for the *Teensy* microcontroller to place it in human-interface-device (HID) mode to act as a USB keyboard. The specific emojis were chosen by the model, adding only the requirement that they must be high-frequency, and at least one must be the heart emoji. Adding many requirements generally reduced design quality – for example, adding the requirement that some of the emojis needed to be at least 5 ASCII characters long was not generally successful (and, the model occasionally generated emojis that were unicode, which is generally easily supported by the USB HID standard, or occasionally generated only single characters instead of full emojis). Similarly, adding the requirement for a short musical tune to play upon pressing an emoji, where the tune should have a similar affect to the emoji (e.g. a love song for the heart emoji, a happy song for the happy emoji, etc.) generally produced only harsh single tones, or the scaffold for generating the music without actual musical tones for each emoji. This scaffold was provided to GPT-4 on its own

in a post-generation step, and the resultant code added to the original code.

B.1 Visible Spectrometer

This code had two central challenges: generating a device driver for an uncommon component (the Hamamatsu C12666MA micro-spectrometer), and using a library for a common component (a display with a common controller). Using different device descriptions, the model either hallucinated non-existent libraries for interfacing to the spectrometer, or generated its own libraries that had the high-level procedure correct (e.g. sending a start pulse to the spectrometer, then continuously sending a clock pulse while reading data using an analog-to-digital converter to read out each of the 256 spectral channels) – though the specifics of the device driver, such as timing or logic levels were typically incorrect and needed to be manually corrected. With respect to the display, three OLED and TFT displays with common display controllers were attempted, and the most successful (the 128x128 OLED using a SSD1351 controller) was used. There were only two small errors in the display code: the initial call to the display had reversed the order of the arguments, and the last call to the display (swapping the backbuffer) was for a different library, and not required here.

B.2 Non-contact temperature sensor

This device consisted of two common components (an MLX90614 non-contact temperature sensor, and an 8-pixel neopixel RGB LED strip). The device generated without issue, and was only modified slightly to reverse the direction the LED bar graph displayed from (to accommodate the mounting constraints of the specific LED strip used).

B.3 Pill alarm assistive device

This device consisted of three common components: an 16x2 LCD with an I2C interface, three hobby servos, and a single pushbutton. The schematics generated largely without issue. The code generally had a number of logic errors that needed correction when the added requirement of oscillating the flags back-and-forth was added, including that the code would oscillate all flags, regardless of which alarm (e.g. pill 1, pill 2, or pill 3) was active. Different generated instances of this device in response to different specifications either kept track of time internally, or used an external realtime clock module for more accurate timekeeping – but all generated devices failed to provide any means of setting the initial time of the device other than manually in code, which is an important usability feature of a clock not explicitly mentioned in the prompt.

B.4 Ultrasonic glasses assistive device

This device generated with only small issues. The device requirements specified using a specific battery-powered ESP32 microcontroller board, but the schematic used digital pin numbers that were unavailable on this specific microcontroller board – these were assigned to other pins trivially. The library the model used for sound generation (TONE) is famously available for most Arduino devices except the ESP32, and was modified to use a different function specific to the ESP32 with a similar signature, but with two added initialization and termination calls. The specific audio frequency range generated by the model was also modified to a reduced range and more fitting for human ears, as the original included high-frequency tones that, while audible, were uncomfortable and resembled a fire alarm.

C Device Descriptions: 25 Benchmark Tasks and 6 Open-generation Devices

Please see the GITHUB repository for a complete set of device descriptions.

D Prompts

Please see the GITHUB repository for a complete set of prompts used in this work.