LogicGlue: Hardware-Independent Embedded Programming Through Platform-Independent Drivers

MANNU LAMBRICHTS, Lancaster University, United Kingdom and Hasselt University, Flanders Make, Belgium
RAF RAMAKERS, Hasselt University, Flanders Make, Belgium
STEVE HODGES, Lancaster University, United Kingdom

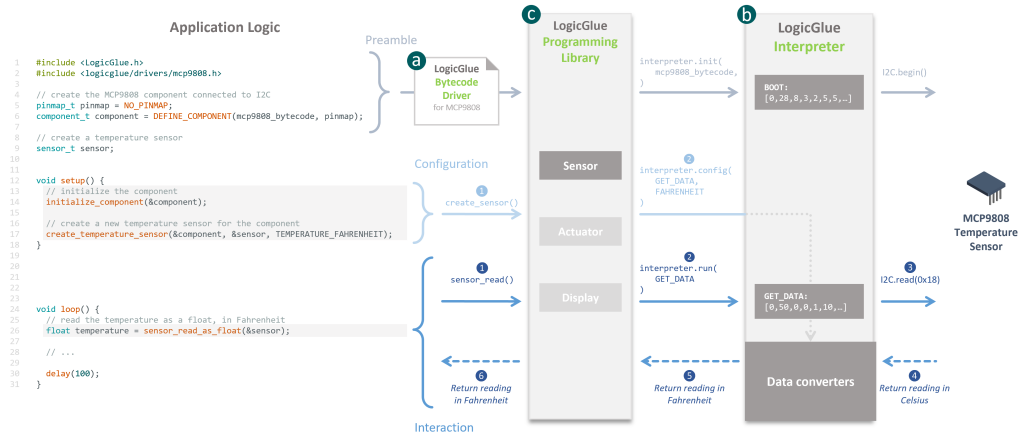


Fig. 1. Overview of LogicGlue. (a) The novel driver specification of LogicGlue encodes the behaviour of drivers in bytecode to ensure platform independence and compatibility across various microcontrollers and programming languages. (b) The LogicGlue interpreter is responsible for processing the bytecode driver specifications and executing platform-specific commands. (c) The LogicGlue programming library is designed to facilitate interaction with electronic components through the interpreter.

The growing capabilities of microcontrollers, sensors, and actuators, coupled with decreasing costs, have led to a proliferation of embedded interactive systems. Prototyping such electronic systems has become democratized across a broad audience, including students, hobbyists, professional engineers, and programmers. Central to this evolution is the ease of software development, and in particular, the availability of low-level drivers and programming libraries which have significantly lowered the barriers to programming these systems. However, this ecosystem often presents challenges due to the tight coupling between programming libraries, drivers, and the underlying sensors and actuators. This frequently leads to compatibility issues. This paper introduces LogicGlue, which addresses these challenges by providing a platform-independent driver specification format. LogicGlue driver specifications allow hardware-independent application logic to be

Authors' Contact Information: Mannu Lambrichts, Lancaster University, School of Computing and Communications, Lancaster, United Kingdom and Hasselt University, Flanders Make, Digital Future Lab, Hasselt, Limburg, Belgium, m.lambrichts@lancaster.ac.uk; Raf Ramakers, Hasselt University, Flanders Make, Digital Future Lab, Hasselt, Limburg, Belgium, raf.ramakers@uhasselt.be; Steve Hodges, Lancaster University, School of Computing and Communications, Lancaster, United Kingdom, steve.hodges@lancaster.ac.uk.

ACM 2573-0142/2025/6-ARTEICS017
https://doi.org/10.1145/3735498

written, facilitating the process of interchanging components with minimal-to-no code adjustments. Unlike existing solutions, LogicGlue supports efficient interfacing via native communication protocols. This approach not only simplifies electronics prototyping but also ensures compatibility between various types of electronic components from different vendors. By reducing the complexity of hardware integration, LogicGlue enables a more seamless exploration of novel interactive behaviours and interfaces, forming a new tool for engineering interactive computing systems.

CCS Concepts: • **Hardware → Embedded systems**; **Sensor devices and platforms**; • **Software and its engineering → Software libraries and repositories**; *Abstraction, modeling and modularity*; • **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; *Interaction devices.*

Additional Key Words and Phrases: Platform-Independent Drivers, Embedded Programming, Software Compatibility, Electronics Prototyping

ACM Reference Format:
Mannu Lambrichts, Raf Ramakers, and Steve Hodges. 2025. LogicGlue: Hardware-Independent Embedded Programming Through Platform-Independent Drivers. *Proc. ACM Hum.-Comput. Interact.* 9, 4, Article EICS017 (June 2025), 46 pages. https://doi.org/10.1145/3735498

1 Introduction

Over the past decade, advances in microcontrollers, sensors, and actuators have integrated embedded systems into many aspects of daily life. At the same time, prototyping embedded systems has become democratized, enabling student [32], hobbyist [4], and professional engineers and programmers to create electronic projects quickly and easily [22]. A key factor in this democratization is the extensibility of microcontrollers with various third-party components and the availability of software libraries, such as the popular Arduino libraries [5], which facilitate programming. Easy iteration of diverse hardware configurations can greatly streamline a user-centred design process and support rapid prototyping of interactive applications, from tangible interfaces to connected IoT environments [21].

Software for embedded systems typically includes three essential elements: low-level drivers, high-level programming libraries, and application logic. Low-level drivers manage direct communication between electronic components and the microcontroller, handling hardware-specific registers and protocols. For example, the SSD1306 OLED display[1] driver controls hardware registers via I2C or SPI protocols to update pixels and adjust settings like brightness. These drivers simplify the complex details of communication protocols and hardware registers, enabling programmers to program application logic using higher-level constructs instead of dealing with intricate timings and specific sequences required for hardware operations. High-level programming libraries provide an additional layer of abstraction, simplifying hardware interaction and making components more accessible to programmers. For instance, the Adafruit_SSD1306 library[2] allows developers to display text on the SSD1306 display without dealing with pixel-level operations.

While low-level drivers and high-level libraries simplify embedded programming, they often come as tightly coupled packages specific to certain components and platforms, limiting compatibility and flexibility. For instance, the DHT sensor high-level library[3] abstracts sensor interaction but is tightly integrated with low-level drivers tailored to the DHT sensor series. This coupling enforces predefined data reading methods and communication protocols, making the library unsuitable for other sensor types like the DS18B20, despite both providing temperature readings in Celsius or Fahrenheit. Additionally, many libraries are platform-specific, such as those for Arduino [4], rendering them incompatible with alternative platforms like Raspberry Pi [39] or micro:bit [7, 32].

[1] https://www.arduino.cc/reference/en/libraries/ssd1306/
[2] https://github.com/adafruit/Adafruit_SSD1306
[3] https://www.arduino.cc/reference/en/libraries/dht-sensor-library/

This fragmentation forces developers to search for or rewrite drivers to ensure compatibility with different hardware and ecosystems.

Some generic libraries, such as the Adafruit GFX Graphics Library[4], improve compatibility with third-party components by abstracting various low-level drivers. However, they often fail to fully leverage hardware-specific capabilities, and extending them requires deep architectural knowledge. For example, Adafruit GFX is primarily designed for SPI-based displays, and additional libraries are needed to support I2C displays like the SSD1306.

Moreover, enhancing these generic libraries with more advanced features is particularly challenging because such additions must remain compatible with a broad range of supported devices. For example, in the GFX graphics library, supporting hardware- or software-based scrolling is difficult due to multiple constraints. Scrolling often relies on hardware acceleration, which is not universally available, requires framebuffer read access that some displays lack, or necessitates modifications to the GFX API—changes that would impact numerous subclasses. The library's maintainers have acknowledged these limitations, noting that the GFX API is largely "set," making significant changes impractical[5]. As a result, while abstraction layers simplify development, they can also restrict access to component-specific features, forcing developers to either accept these limitations or implement complex workarounds.

High-level communication protocols like Jacdac [14] and CoAP [37] offer an alternative approach by abstracting low-level drivers into standardized interfaces. Jacdac, for instance, uses services to separate application logic from low-level drivers, increasing compatibility and significantly lowering the barrier for embedded programming. However, since electronic components cannot communicate directly with these high-level protocols, an additional microcontroller is required for each electronic component to handle the conversion, introducing latency and potential loss of unique component features that are not captured by the protocol.

In this paper, we introduce LogicGlue, a novel software stack that supports platform-independent drivers and, as such, allows hardware-independent application logic to be written. To realize this, LogicGlue consists of a novel driver specification (Figure 1(a) for expressing the behaviour of electronic components. The LogicGlue driver specification defines the complete functionality and characteristics of an electronic component, ranging from specific communication protocols to data formats and units. As these definitions are specified in bytecode, the LogicGlue driver specifications can be processed by any microcontroller and any programming language, making them platform-independent. As such, users are not restricted to a single microcontroller or platform.

The LogicGlue interpreter (Figure 1(b) runs on the system's microcontroller and translates the instructions from the driver specification into platform-specific commands. The high-level LogicGlue programming library (Figure 1(c) further facilitates interfacing with electronic components via the interpreter. Unlike high-level communication protocols such as Jacdac [14], LogicGlue does not require the translation of features of electronic components to services nor the conversion of communication messages to a single protocol. Instead, the driver commands of LogicGlue ensure all components' features remain available, and interfacing is done via the native signals supported by the electronic components, avoiding latency.

LogicGlue empowers developers to write hardware-independent application logic. Swapping, for example, a temperature sensor working over the I2C protocol, with one outputting analog voltage readings, does not require rewriting application logic as LogicGlue automatically handles the required data conversions. Furthermore, our buildup ensures that microcontrollers or development platforms that implement the LogicGlue interpreter and the LogicGlue programming library are

[4]https://learn.adafruit.com/adafruit-gfx-graphics-library/overview
[5]https://github.com/adafruit/Adafruit-GFX-Library

instantly compatible with all electronic components for which LogicGlue driver specifications are available. Likewise, electronic components for which a LogicGlue driver specification is available are compatible with any microcontroller that supports LogicGlue. By abstracting hardware-specific details, LogicGlue allows developers to focus on prototyping and refining interactivity rather than managing low-level communication and compatibility issues. This aligns with the findings of Raffaillac and Huot [42], which highlight the importance of development tools that reduce hardware-related complexities in interactive system research.

In summary, we contribute:

(1) A LogicGlue driver specification format for defining how electronic components behave—including protocol details, data formatting, and parameter units—expressed as a portable bytecode.
(2) A LogicGlue interpreter that executes these bytecode-based driver specifications on a microcontroller, translating abstract instructions into platform-specific commands.
(3) A high-level programming library that helps developers to interface with the interpreter, preserving access to all component features while using the components' native communication protocols.
(4) An optional block-based programming interface that simplifies the creation of driver specifications by assembling bytecode visually rather than using text.

2 LogicGlue

LogicGlue is a software abstraction layer designed to change how electronic components, such as sensors, actuators, and displays, are interfaced at the hardware level. It introduces a platform-independent driver format that enables consistent initialization, communication, and data handling across diverse microcontroller platforms and component types. LogicGlue does not process interaction-level events such as mouse clicks, gestures, or key mappings. Instead, it operates at a lower level, transferring data with a component in its native format and making it available to application logic through a unified interface.

This positions LogicGlue as an infrastructural layer within the prototyping stack: it bridges the gap between the raw hardware interface – often fragmented and vendor-specific – and the high-level behaviours that developers typically want to use, such as providing visual feedback, sending network messages, or responding to user input. By decoupling hardware access from behaviour, LogicGlue allows developers to iterate on hardware choices without needing to modify their application code. This is especially critical in early-stage prototyping, adaptive systems, and exploratory interaction design, where hardware configurations frequently change as ideas are tested and refined.

Consider, for instance, a physical button used in a custom-built mouse. In a traditional system, replacing that button with a capacitive touchpad or proximity sensor might require rewriting the low-level code that detects input, interprets signal thresholds, and triggers events. With LogicGlue, the application can continue to treat the input as a binary signal – "pressed" or "not pressed" – regardless of whether it comes from a mechanical switch, a capacitive sensor, or even an optical trigger. The developer simply swaps the LogicGlue driver; the logic that generates the USB click remains unchanged.

This hardware-independence becomes particularly valuable in public or adaptive installations. In a museum exhibit, for example, the original interaction may rely on a push-button physically embedded into an exhibit. If that button proves fragile or inaccessible to some visitors, it could be replaced with a pressure sensor, a capacitive plate, or a gesture-detecting sensor – each using different protocols and signal types. Yet, the media playback logic, animation triggers, and exhibit

control systems remain untouched because LogicGlue abstracts the interaction as a consistent binary input.

In smart-home interfaces, the benefits of hardware abstraction extend to sensors with varied performance characteristics. Developers might alternate between analog and digital temperature sensors, or swap light sensors with differing sampling rates or output formats. LogicGlue enables this flexibility without requiring changes to the code that evaluates conditions (e.g., "turn on fan if temperature > 25℃") or updates to display elements. The application logic remains focused on what to do with the data, not how to extract or interpret it. In handheld or wearable devices, developers often face trade-offs between display size, resolution, power consumption, and interface protocols. LogicGlue allows them to replace, for example, an I2C OLED display with one that uses SPI, while preserving the rendering logic that populates the screen with text or status icons. This simplifies experimentation with the form factor and hardware integration without entangling it with the specific behaviour or interface logic of the device.

In all these cases, LogicGlue serves as a layer of abstraction that insulates application logic from hardware variability. It empowers developers to adapt, iterate, and refine their systems rapidly, whether for prototyping, deployment in the field, or personalization, while ensuring that functionality remains intact as the physical components evolve.

2.1 Writing Application Logic

To explain the procedure for writing the application logic, we will provide an example from the perspective of a DIY enthusiast, Alex, who integrates a temperature sensor into a prototype with the help of LogicGlue. Considering the wide variety of temperature sensors available, such as analog, digital, and infrared temperature sensors, Alex wants to experiment with different models to find the most suitable one for his project. Traditionally, this would be a lengthy and complex process, as each component might use a different protocol and require different signal processing methods. In the best-case scenario, a software library is available for a component, and Alex only needs to rewrite the application logic to integrate the library. For example, the library of the MCP9808 digital temperature sensor[6] embeds features for initializing and using the I2C protocol, while the library for an analog temperature sensor handles all analog-to-digital conversions. If no library is available that is compatible with the development platform used, an additional driver has to be written first based on the specifications in the datasheet. Figure 2 shows the major differences in code for interacting with the MCP9808 digital temperature sensor over I2C, using the Adafruit library, and the TMP36 analog temperature sensor using analog readings, on the Arduino platform. Besides differences in protocols between the sensors, there are also major differences in output readings, as one temperature sensor outputs temperatures in Celsius while the other outputs a voltage. Additional processing is thus needed.

Using LogicGlue, Alex does not need to write multiple versions of the application logic to test different temperature sensors. Instead, he can include the respective driver specification file and test each temperature sensor using the same application logic code. Figure 3(c) shows the LogicGlue code to interact with the MCP9808 digital temperature sensor working over I2C, and the TMP36 analog temperature sensor using analog readings. As shown in Figures 3(a) and (b), only the preamble changes when swapping between the two very different temperature sensors. These changes include the driver specification file, the parameter passed in the constructor, and the references to the microcontroller pins that the component is connected to. Despite the difference in output readings (Celsius versus voltages), Alex specifies in both code snippets that he prefers

[6] https://www.arduino.cc/reference/en/libraries/mcp9808/

a TMP36 temperature sensor (analog)

```
#define TMP36pin A0

void setup() {
}

void loop() {
  // read the temperature
  int sensorValue = analogRead(TMP36pin);

  // convert the reading into Fahrenheit
  float voltage_mV = (((sensorValue * 5.0) / 1023.0) - 0.5) * 1000;
  float temperature_C = voltage_mV / 10.0;
  float temperature_F = (temperature_C * 1.8) + 32;

  // ...

  delay(100);
}
```

b MCP9808 temperature sensor (digital)

```
#include <Wire.h>
#include "Adafruit_MCP9808.h"

Adafruit_MCP9808 tempsensor = Adafruit_MCP9808();

void setup() {
  // initialize the sensor and setup I2C
  if (!tempsensor.begin(0x18)) {
    return;
  }
}

void loop() {
  // read the temperature
  float temperature_C = tempsensor.readTempC();

  // convert the reading into Fahrenheit
  float temperature_F = (temperature_C * 1.8) + 32;

  // ...

  delay(100);
}
```

Fig. 2. The traditional application code that needs to be written to interact with a) the TMP36 analog temperature sensor and b) the MCP9808 digital temperature sensor.

temperature readings in Fahrenheit (line 20). The LogicGlue interpreter automatically converts the readings into the format Alex prefers.

a TMP36 temperature sensor (analog)

```
#include <LogicGlue.h>
#include <logicglue/drivers/tmp36.h>

// create the TMP36 component connected to pin A0
pinmap_t pinmap = DEFINE_PINMAP(A0);
component_t component = DEFINE_COMPONENT(tmp36_bytecode, pinmap);
```

b MCP9808 temperature sensor (digital)

```
#include <LogicGlue.h>
#include <logicglue/drivers/mcp9808.h>

// create the MCP9808 component connected to I2C
pinmap_t pinmap = NO_PINMAP;
component_t component = DEFINE_COMPONENT(mcp9808_bytecode, pinmap);
```

c

```
// create a temperature sensor
sensor_t sensor;

void setup() {
  // initialize the component
  initialize_component(&component);

  // create a new temperature sensor for the component
  create_temperature_sensor(&component, &sensor, TEMPERATURE_FAHRENHEIT);
}

void loop() {
  // read the temperature as a float, in Fahrenheit
  float temperature = sensor_read_as_float(&sensor);

  // ...

  delay(100);
}
```

Fig. 3. a) Preamble for including the LogicGlue driver specification for the TMP36 analog temperature sensor. b) Preamble for including the LogicGlue driver specification for the MCP9808 digital temperature sensor. c) Application logic interacting with either temperature sensor using temperatures in Fahrenheit.

2.2 Writing Driver Specifications

LogicGlue driver specifications include all the functionality for driving an electronic component. To streamline writing driver specifications, we developed a block-based specification interface based on Blockly[7]. As shown in Figure 4, users compose commands by selecting command blocks representing various functionalities. When the specifications are complete, the interface stores this

[7]https://developers.google.com/blockly

graphical representation of the driver specifications as a bytecode sequence in a header file. This file is then included in the application logic code as demonstrated in Section 2.1. To correctly compose driver specifications, a good understanding of the component's datasheet is required. However, this is a one-time effort, best done by component manufacturers or experienced engineers, and then shared with all customers or users.

It is important to emphasize that the block-based interface is merely a convenient tool for writing driver specifications — it does not define the LogicGlue driver specification language itself. The actual specification is written in bytecode, which is processed and executed by the LogicGlue interpreter on the microcontroller. The block-based interface abstracts the complexities of writing bytecode manually, embedding constraints such as required parameters for each instruction. However, it does not introduce novel contributions in visual programming paradigms, nor does it influence LogicGlue beyond aiding in the initial creation of driver specifications.

Fig. 4. LogicGlue's graphical interface for creating drivers using the driver specification.

The following example illustrates composing the driver specifications for the MCP9808 temperature sensor[8]. In the *init* procedure, we initialize the component's communication protocol, which, in this case, is I2C. As shown in Figure 5, we insert a *configure* block (line 1), select I2C as the protocol, and set the frequency to 400kHz as specified in the sensor's datasheet. The datasheet further details that this component is available via address 0x18 on the I2C bus, and uses 16-bit registers (line 2). A common practice for I2C devices then involves verifying the component's presence by reading out the manufacturer ID register (0x06) and comparing it to the ID detailed in the datasheet (0x54) using an assert (line 3). The last block in the start procedure involves setting the sensor's default configuration, using a write command, to ensure the sensor uses its default settings at startup. As shown in line 4, we set the data of the configuration register (0x01) to its default value (0) as specified in the sensor's datasheet.

[8]Datasheet: https://ww1.microchip.com/downloads/en/DeviceDoc/25095A.pdf

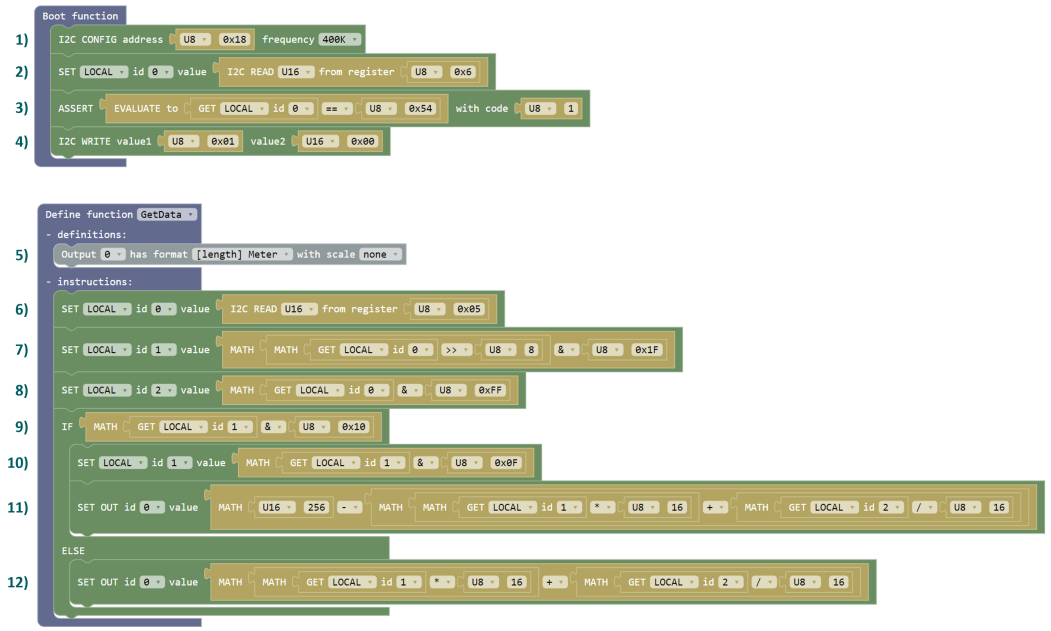


Fig. 5. LogicGlue driver specification for the MCP9808 temperature sensor.

The next step involves defining the features supported by the MCP9808 temperature sensor, such as temperature readings, resolution updates, and sleep and wake-up functionalities. Each functionality is created using a *functionality* block. For specifying the temperature reading functionality, we select *GetData* as the function category. This field defines the primary functionality of the component, compared to, for example, the *resolution* feature. Within the definitions of this block, we specify the return parameter type and instructions for reading the temperature. The return type is set using a *output* block (line 5), which characterizes the return type. As specified in the datasheet, the MCP9808 temperature sensor returns temperature readings in Celsius. Alternatively, this block can support other function categories, such as setting data, triggering specific features like sending the pixel buffer to a display, or reading internal parameters like the internal temperature of an electronic component.

To read the temperature (line 6), we follow the instructions specified in the sensor's datasheet. Using a *read* block, we first read the temperature register (0x5) from the MCP9808 temperature sensor and store the data in a temporary variable. As the data is in two's complement format, we follow the calculations in the datasheet to convert the bits to a decimal temperature. First, we split the 16-bit value into upper and lower bytes (lines 7 and 8) and clear the flag bits in the upper byte. As the positive and negative temperature data are computed differently, we use an *if* block (lines 9-12) to check if the sign bit (0x10) indicates a negative value (line 9). If this bit is set, the if-test evaluates to true, and lines 10 and 11 are executed. Line 10 resets the sign bit, while line 11 executes the mathematical operations specified in the datasheet to calculate a negative temperature in Celsius and stores the result in the output variable. If the bit indicating a negative value is not set, line 12 is executed, which calculates and stores a positive temperature in Celsius.

Once all the functionality is defined the LogicGlue interface automatically stores this driver specification as a bytecode sequence in a header file.

3 Related Work

This section discusses the contributions that have influenced the development of LogicGlue, focusing on advancements in software abstraction, standardized communication interfaces, and intermediate representation layers. Detailed descriptions of all LogicGlue instructions and their semantics are provided in Appendix A.

3.1 Software Abstraction

Over the years, embedded systems have been significantly shaped by the advent of software platforms and frameworks aimed at abstracting hardware complexities. Notable platforms such as PlatformIO [40], Mbed OS [31], and Zephyr [46] have been instrumental in offering operating system-like functionalities to microcontrollers. These platforms mask the intricacies of hardware, allowing developers to concentrate on application logic. Easy-to-use hardware and software platforms such as Arduino [4] and micro:bit [32] have also played an important role in democratizing embedded programming for a broad audience.

Real-time operating systems (RTOSes) like FreeRTOS [17], provide concise, scalable, and flexible software management for embedded devices. Similarly, TinyOS [28] has played a pivotal role in promoting the development of networked sensor systems, highlighting the importance of specialized platforms in the advancement of IoT and embedded applications.

Despite these advances, challenges persist in integrating external hardware components. Zephyr's implementation of device drivers [45] provides a framework for hardware management, yet adapting these drivers for new components requires a deep understanding of hardware-software interaction. While Zephyr's device drivers share a concept similar to the LogicGlue driver specification, LogicGlue focuses on interacting with external hardware. At the same time, Zephyr is designed to offer a standardized approach for setting up hardware peripherals. Furthermore, LogicGlue offers a convenient block-based interface for creating the drivers.

In parallel, frameworks like CODAL [13] and Arduino [4] have significantly eased microcontroller programming, streamlining direct hardware interaction. However, they often fall short in addressing the complexities of integrating external components, a task that still poses considerable challenges. Libraries such as the Adafruit GFX Graphics library [12] aim to bridge this gap by abstracting hardware communication. But these libraries typcially can't leverage the unique features of each component, and integrating them in the application logic requires a good understanding of the functionality offered by the library.

Integrated development environments (IDEs) have made significant inroads in addressing these challenges. The Arduino IDE [6], DeviceScript [33], and Microsoft MakeCode [9] integrate tools that facilitate the discovery, selection, and integration of software libraries, enhancing the efficiency of the development workflow. These IDEs, alongside advanced environments like Visual Studio Code [34] with its rich features for embedded development, play a pivotal role in lowering the entry barriers to embedded programming.

In general, while existing software platforms and frameworks have significantly simplified working with hardware, they often come with a standardized abstraction layer that doesn't account for the unique functionalities of individual components or the varied needs of developers. LogicGlue introduces a novel approach by offering platform-independent driver specification alongside a versatile programming library, ensuring developers can write hardware-independent application logic. This eliminates the limitations posed by the tight coupling of drivers and libraries in conventional systems, allowing for seamless hardware changes without extensive code adjustments.

Beyond enabling hardware-independence, LogicGlue also provides a foundation upon which higher-level interaction frameworks, such as ICON [15], can be built. ICON-like systems address the

remapping of application-level input semantics, such as assigning mouse actions to keyboard keys or redirecting gestures across contexts, operating at a layer above the hardware. While LogicGlue does not influence how input is interpreted at the application level, it ensures consistent, low-level communication between diverse components and the user application. This makes it an ideal substrate for systems like ICON by providing uniform access to and reconfiguration of a broad range of input and output hardware. Together, this complementary layering would enable the development of modular, adaptable interactive systems, where both hardware connectivity and user interaction logic can be flexibly redefined.

In this broader landscape of input abstraction, prior work by Accot et al. has introduced formal models such as transducers to describe how physical devices mediate between user actions and system responses [1, 2]. These contributions provide a valuable theoretical foundation for understanding device behaviour in terms of structured state and signal transitions. While LogicGlue does not directly implement such formal transducer models, its architecture – by exposing structured, platform-independent access to device-level behaviour – offers a practical foundation upon which such models could be realized or operationalized in real-world systems.

3.2 Standardized Communication Interfaces

The concept of integrated modular systems has seen substantial development, with ecosystems like Jacdac [14], Modular-Things [43], and .NET Gadgeteer [20] leading the way in standardized communication interfaces. These systems offer a range of compatible components that communicate through standardized protocols, enabling easy system assembly and expansion. The Raspberry Pi platform [39], with its extensive ecosystem of hardware add-ons and HATs (Hardware Attached on Top), exemplifies the power of modular design in promoting system scalability and interoperability.

SoftMod [25], a concept that emphasizes configuring component behaviour through their physical arrangement, represents a novel approach in this domain. This strategy facilitates a tangible and intuitive method for modifying system functionalities, showcasing the potential for physical configuration to impact software behaviour directly. Further contributions to this field include the Intel Edison [16] and PMod [41] platforms, which were designed to foster innovation in IoT and embedded projects through modular components and standardized interfaces. Similarly, the BeagleBone [11] series offers an open-source platform that encourages experimentation and development with its cape plug-in boards, underlining the versatility of modular design in embedded systems.

Interoperability in embedded systems necessitates interacting over diverse communication protocols like SPI, I2C, and UART. While these protocols are frequently used to interact with electronic components, using them requires significant knowledge in embedded development [23]. High-level protocols such as Jacdac [14] provide a simplified method for device communication, wrapping low-level communication protocols like I2C and SPI into a high-level communication standard. In practice, they introduce an ecosystem of modules that all share the same communication interface to streamline system assembly. In addition, high-level protocols tailored for IoT applications, like MQTT [38] and CoAP [37], have become prominent, offering lightweight solutions for resource-constrained environments. These protocols exemplify advances in ensuring devices from various manufacturers can communicate seamlessly, fostering a more cohesive ecosystem. Here, frameworks like TinyOS [28] and Static TypeScript [10] have made contributions by focusing on networked systems and providing platforms for building complex, interconnected devices. However, compared to low-level communication protocols like SPI and I2C, these approaches often introduce additional translation steps to make electronic components compatible with the communication interface. In particular, each feature of an electronic component needs to be able to be exposed

through the standardized interface, potentially leading to the loss of unique features and latency issues.

In contrast, high-level libraries such as the Adafruit CircuitPython Register library [3] provide a structured approach to interfacing with hardware registers. This library utilizes data descriptors, which act as Python attributes, allowing developers to define device drivers in an intuitive manner. By encapsulating I2C and SPI register access within these descriptors, the library abstracts low-level communication while maintaining direct hardware interaction. While such approaches simplify device driver development, they are inherently limited to register-based communication protocols, such as I2C and SPI, and cannot be used for components that rely on analog signals, digital GPIO interactions, or pulse-based communication. This restricts their applicability to a subset of electronic components. LogicGlue extends this concept by offering a platform-independent driver specification that supports a broader range of hardware interfaces, ensuring that hardware interactions remain decoupled from language-specific implementations and enabling seamless integration across different microcontrollers and development environments.

LogicGlue stands out by supporting direct interfacing with hardware components through their native protocols without resorting to standardized interfaces. This ensures the unique features of each component are preserved, offering developers a more efficient and feature-rich integration experience. Furthermore, the platform-independent nature of LogicGlue driver specifications ensures that any microcontroller or development platform implementing the LogicGlue interpreter and programming library becomes instantly compatible with all supported electronic components. Developers can switch between components with different communication protocols or functionalities without rewriting application logic, significantly reducing complexity and improving the adaptability of embedded systems development. This approach not only streamlines development but also enhances the potential for creative hardware solutions by simplifying the integration of diverse components.

3.3 Intermediate Representation Layers

The adoption of high-level programming languages and abstraction layers has significantly transformed software development practices. Platforms like Node-RED [36], which provides a visual programming environment for IoT applications, illustrate the effectiveness of abstracting complex code into more accessible formats. Similarly, the introduction of TypeScript [10] has offered developers a powerful tool for building large-scale applications by providing types and high-level syntax that compile down to JavaScript, suitable for web and server environments.

An alternative approach to cross-platform embedded software is to execute high-level code on microcontrollers using lightweight virtual machines (VMs) or interpreters. Early examples such as Maté for TinyOS [27] demonstrated this in sensor networks, while modern systems like MicroPython and CircuitPython [18] embed Python interpreters to support rapid prototyping with access to hardware features. These VMs offer uniform abstractions and enable dynamic code updates, though they introduce memory and processing overhead [30]. To address this, bytecode-based systems like OMicroB [44] and WARDuino [19] compile high-level code into efficient, portable representations. WARDuino, for instance, adapts WebAssembly to embedded use, achieving significantly improved performance over interpreted JavaScript. Offloading dynamic compilation to host systems further enhances runtime efficiency on microcontrollers [35]. Additionally, frameworks employing design patterns can streamline application development across diverse hardware platforms [8]. Collectively, these systems illustrate that well-designed interpreters can balance portability and runtime efficiency, though they require careful handling of debugging and memory management in constrained environments [30].

Intermediate representation layers (IRL) are key in bridging high-level programming constructs with the lower-level code required by microcontrollers and embedded devices. For instance, LLVM [29] provides a wide range of tools and libraries that support converting high-level language code into machine code, facilitating cross-platform application development. This concept is crucial in understanding how abstract code structures can be effectively translated into executable commands that run on hardware devices.

Traditional software development practices for embedded systems often rely on converting the whole application logic into an IRL, focusing primarily on programming the microcontrollers themselves. This approach is instrumental in bridging the gap between high-level programming constructs and the lower-level executable code required by microcontrollers, as seen in platforms leveraging LLVM [29] or similar technologies. The primary objective here is to streamline the development process for the microcontroller's software, ensuring that high-level abstractions are effectively translated into machine-level commands that the hardware can execute.

LogicGlue, while embracing a conceptually similar use of IRLs, diverges in its application and objectives. Rather than focusing on the microcontroller's entire application logic, LogicGlue facilitates the interactions with hardware components. Its driver specification outlines the commands for interfacing with hardware components. This specificity ensures that developers can engage with the unique functionalities of each component directly, without the intermediary step of translating general-purpose application logic into hardware-specific commands.

4 LogicGlue Driver Specification

The LogicGlue driver specification provides a comprehensive framework to characterize the complete functionality of an electronic component. This ranges from initialization procedures to read and write actions, as well as fine-tuning the component's settings. The LogicGlue driver specification is based on the RISC (reduced instruction set computer) architecture and is Turing complete, ensuring it can fully express and execute any existing component drivers, given sufficient resources. LogicGlue supports the full spectrum of low-level operations, including data handling, conditional execution, hardware communication protocols, and program flow control. As a result, LogicGlue provides a universal abstraction layer that eliminates the need for hardware-specific code, enabling seamless interoperability across different microcontrollers and embedded platforms. By focusing on platform-independent, bytecode-based specifications, LogicGlue ensures that developers can engineer the interactive behaviours of embedded systems without becoming entangled in each component's low-level specifications.

Each instruction in the LogicGlue driver specification is represented by a predefined numeric code, organized within an enumeration (enum). Using an enum allows each instruction to be given a descriptive name, which enhances code readability and maintainability. While the driver specification itself is physically stored as a traditional array of bytes, this section presents all instructions using their enum names, along with indentation and colour coding, to improve clarity and comprehension. Additionally, we have included a side-by-side visualization of the optional block-based representation in the LogicGlue interface to further aid understanding. For complete examples demonstrating how these instructions are applied in practice, we refer the reader to Appendices E, F, and G.

4.1 Function Definitions

The LogicGlue driver specification comprises several functions that represent the available features of an electronic component. This section explains how these functions are defined within the LogicGlue driver specification, using the driver for the HC-SR04 ultrasonic distance sensor as an example (Figure 6).

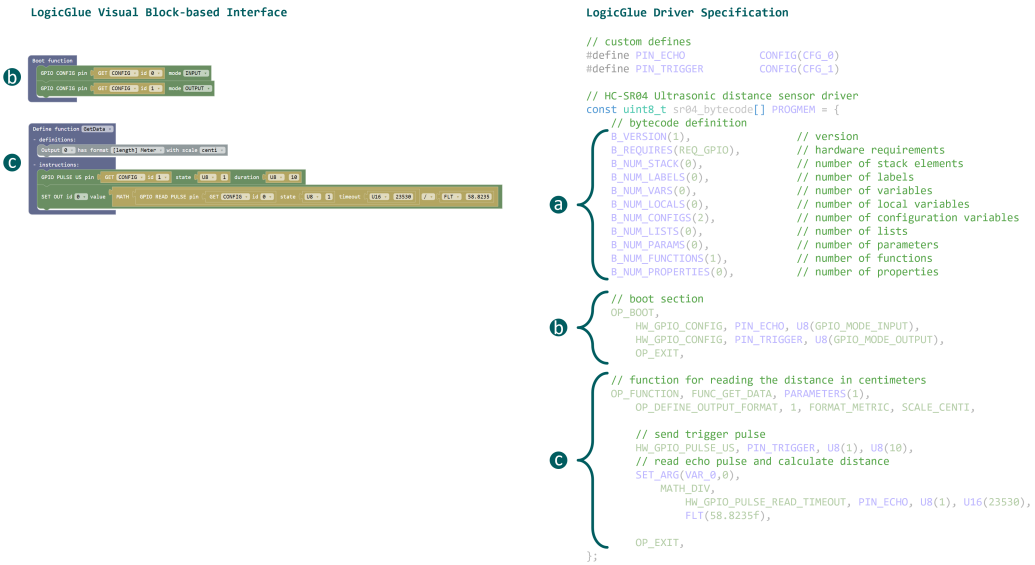


Fig. 6. Driver specification for the HC-SR04 ultrasonic distance sensor. a) shows the bytecode definition, b) the boot function, and c) contains all function definitions.

The specification begins with the bytecode definition (Figure 6(a)), specifying the required microcontroller resources for the environment in which the driver will run, including communication protocols and memory allocation requirements. These details ensure that the electronic component is compatible with the intended microcontroller or platform, which is crucial for correct operation within a given hardware setup. For example, if a sensor requires an I2C communication protocol and specific GPIO pins, these requirements are explicitly stated to prevent issues related to incompatibility. When using the LogicGlue visual interface, the bytecode definition is automatically determined and added when storing the driver specification header file.

Following the bytecode definition, the boot function (Figure 6(b)) provides instructions necessary for configuring both the microcontroller and the electronic component. This includes configuring communication protocols, initializing GPIO pins, and setting configuration registers. For instance, configuring an I2C temperature sensor would involve setting the I2C address and preparing the necessary registers to read temperature data. These steps ensure the electronic component is correctly initialized and ready for operation. The boot section is executed automatically.

The next section of the LogicGlue driver specification defines the functions of the component (Figure 6(c)). Each function is described by its name and the format of its input and output parameters, followed by a series of instructions that detail the steps required to perform the function. The function name indicates the type of functionality, such as *light sensing*, *temperature reading*, or *display output*. Parameters are characterized by the format name and a scale, which allows for adjusting the data value according to predefined standards such as metric scales. These details are crucial for LogicGlue's automated data conversion (section 5.2) and, ultimately, allow for swapping between electronic components without rewriting application logic. For instance, if Fahrenheit is used in the application logic, LogicGlue offers readings in this unit regardless of whether the temperature sensor returns readings as Fahrenheit, Celsius, or Kelvin.

Like function overloading in traditional programming, LogicGlue supports multiple alternative implementations of similar functionalities that are differentiated by their input or output parameters. This is useful when, for example, defining driver instructions for the MPU-6050 accelerometer[9] which has built-in support to output quaternions and Euler angles. To determine the best driver function, LogicGlue prioritizes functions based on the number of required data converters to match the data format used in the application logic.

In addition to functions, the driver specification also details the properties of the electronic component, such as display size, gain, and sensor integration time. Properties are managed similarly to functions but do not require data format conversions for input and output values. Instead, properties are defined by a name and the range of values they accept, which can be a predefined set (e.g., sensor gain) or numeric values (e.g., display size). LogicGlue automatically verifies the compatibility of a given value with a property. For example, when setting the gain of a sensor, LogicGlue supports various generic gain settings of 1x, 2x, 4x, 8x, and so on. However, the TCS34725 color sensor[10] specifically only supports gain settings of 1x, 4x, 16x, or 64x. By defining the property with a set of accepted values, as illustrated in Figure 7, LogicGlue ensures that only compatible gain settings are accepted, maintaining consistent behaviour when swapping between different electronic components and showing an error in case incompatible values are used.

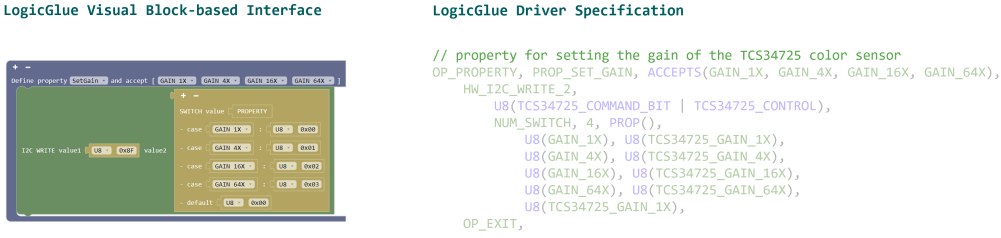


Fig. 7. Defining a property for setting the gain of the TCS34725 colour sensor.

Finally, properties can be defined as static when they return a constant value, providing a straightforward way to access static information about the component. For example, Figure 8 shows the definition of the static properties for getting the display size of the SSD1306 OLED display, which will never change and thus can be defined as static.

Fig. 8. Defining static properties for getting the size of the SSD1306 OLED display.

[9] https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/
[10] https://ams-osram.com/products/sensors/ambient-light-color-spectral-proximity-sensors/ams-tcs34725-color-sensor

4.2 Numeric Instructions

LogicGlue features a versatile subsystem for handling numeric instructions. A numeric instruction is an instruction that evaluates to a numeric value, such as an integer or floating-point number. These instructions can represent constant values, the results of mathematical operations like addition and subtraction, or values received from hardware peripherals like GPIO pins or communication protocols. For each numeric instruction, LogicGlue keeps track of its data type and supports unsigned integers (U8, U16, U32), signed integers (I8, I16, I32), and numbers in both floating (FLT) and fixed (FIX) point formats. Detailed descriptions of all numeric values and their specific semantics are included in Appendix B.

A key feature of the numeric subsystem is its ability to use numeric instructions as inputs for other numeric instructions, enabling the nesting of operations. This allows for the creation of complex numerical expressions and operations. For example, Figure 9 illustrates various numeric operations: a) the sum of two integers, b) multiple nested mathematical operations, c) the state of a GPIO pin, and d) an inline if-test.

It is important to note that since the LogicGlue driver specification is stored as an array of unsigned bytes, signed numbers must be cast, and large integers or floating-point numbers must be split across multiple bytes. LogicGlue provides convenient macros to handle these conversions automatically.

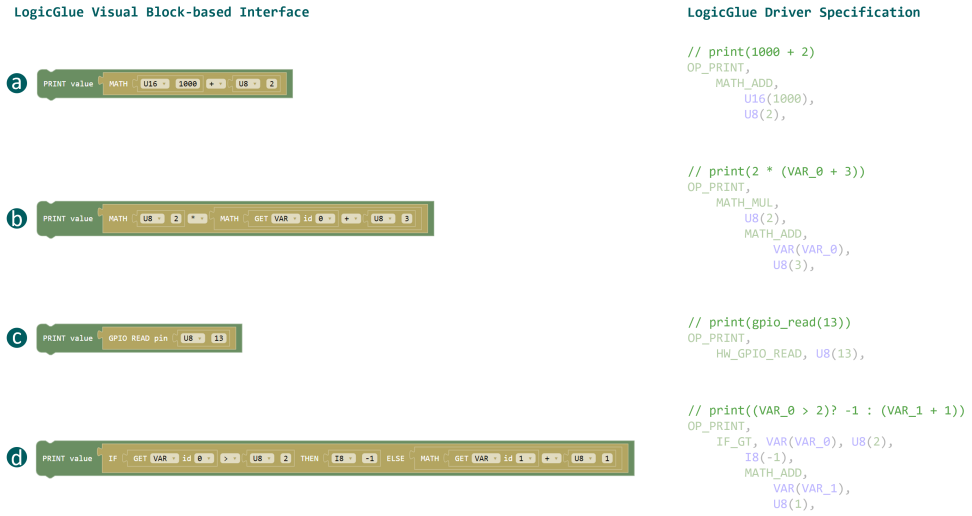


Fig. 9. Illustration of the numeric subsystem within the LogicGlue driver specification, demonstrating various numeric operations.

4.3 List Instructions

List instructions in LogicGlue function similarly to numeric instructions, providing a flexible way to handle arrays of data. Like numeric instructions, LogicGlue maintains the data type of each list to ensure type consistency. Lists are commonly used as data buffers in drivers for displays or as a convenient method for sending multiple bytes over a communication protocol. For example, Figure 10 shows an example of the instructions for sending the pixel buffer to the SSD1306 OLED

display using a list for the data commands (line 2) and a list for the pixel buffer (line 4). Detailed descriptions of all list items and their specific semantics are included in Appendix C.

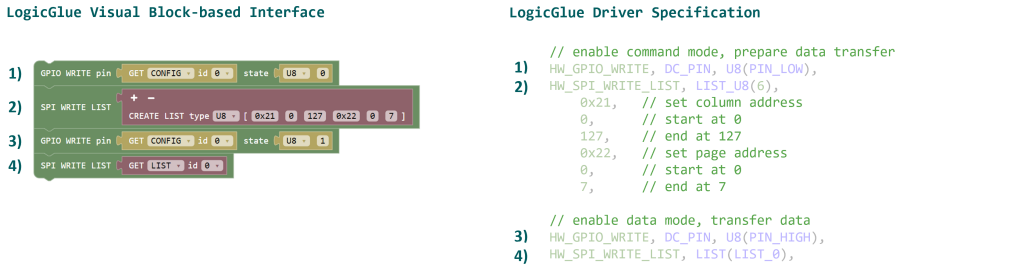


Fig. 10. Illustration of the list subsystem within the LogicGlue driver specification, demonstrating the instructions for sending the pixel buffer to the SSD1306 OLED display.

LogicGlue supports various list types, including integer lists, floating-point lists, and binary arrays. Binary arrays are a special type of array where 8 bits are grouped and stored as a regular byte but can be individually addressed. LogicGlue supports two variants of binary arrays: one where the 8 bits in the x-direction are combined into one byte, and another where the 8 bits in the y-direction are combined. Binary arrays are typically used for single-colour displays like the SSD1306 and dot-matrix displays, where each pixel can either be on or off.

When developing a LogicGlue driver for displays like the SSD1306 OLED, it is necessary to define functions that set the colour of pixels based on specified x and y coordinates. Typically, this involves updating the pixel colour by writing a new value to the internal pixel buffer. While straightforward, this method becomes inefficient when updating a large number of pixels in a loop because each iteration requires evaluating the input values and executing update instructions, leading to significant computational overhead.

To address this inefficiency, LogicGlue introduces specialized function types that execute predefined actions more effectively. For example, the *OP_DEFINE_FUNCTION_TYPE* instruction allows developers to define optimized driver functions, such as *SET_LIST_LOOP*, which streamlines the process of updating multiple values in a list. This function type enables developers to specify a range of indices to be updated at once and to provide either a static value or a callback function that determines the new value for each pixel. By employing an optimized loop within the LogicGlue interpreter, the *SET_LIST_LOOP* function significantly reduces the computational load by minimizing the repetitive evaluation of values. This efficient approach ensures rapid updates while maintaining the capability to automatically convert values as necessary, enhancing the overall performance and responsiveness of the display updates. As demonstrated in Section 7, updating the values for the SSD1306 OLED display is as efficient as traditional approaches.

4.4 Branching Instructions

LogicGlue offers a range of instructions that facilitate complex control flows, similar to traditional programming languages. For instance, the *IF*, *IF_ELSE*, and *IF_ELIF_ELSE* instructions allow for conditional branching, akin to their counterparts in conventional programming. The *LOOP* instruction requires start, end, and increment values, executing the subsequent instruction multiple times based on these parameters. Additionally, the *FOREACH* and *FOREACH_BYTE* instructions iterate over the items in a list, using each list item or byte, respectively. While *FOREACH* returns a single item, *FOREACH_BYTE* is particularly useful for binary arrays or lists storing 16- or 32-bit values,

as it returns each byte separately, regardless of the list type. For example, in Figure 11, the driver for the dot-matrix display uses a binary array as a pixel buffer and requires the row address to be sent before each data byte. Using the *FOREACH_BYTE* instruction, this operation becomes more efficient as it allows the driver to iterate through each byte of the binary array, sending the row address and the corresponding data byte in a streamlined manner.

Fig. 11. Example of the *FOREACH_BYTE* instruction.

LogicGlue supports various methods for branching the program flow. Labels act as designated jump points within the LogicGlue driver specification, with the *CALL* instruction leveraging these labels to dynamically direct the program counter. The *CALL* instruction functions similarly to traditional programming functions, supporting arguments and creating a separate environment for the function that is called. The *RETURN* instruction reverts the flow of execution back to its original position before the jump, effectively managing the program's execution stack.

In addition to control flow, LogicGlue driver specification supports various types of data storage and manipulation, distinguishing between global and local variable scopes. Global variables are persistent and accessible throughout the entire driver specification, while local variables are temporary and only exist within specific functions, managed via a stack mechanism. This stack-based approach allows for the dynamic allocation and deallocation of local variables as functions are called and returned.

When functions are invoked, arguments are passed by reference, linking them to global variables. This method allows functions to alter different variables by updating the references passed as arguments, enabling reusable functions to operate on varying data without redundancy. Figure 12 illustrates the different scopes of variables within the LogicGlue driver specification.

4.5 Advanced Instructions

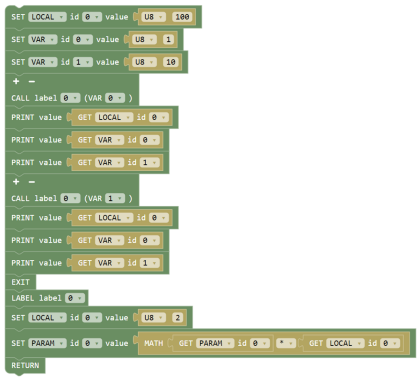
In addition to using labels as jump points for function calls, labels can also be utilized with the *GOTO* instruction, which moves the program counter while maintaining the current context. Beyond the basic *GOTO* instruction, LogicGlue driver specification includes conditional instructions such as *GOTO_IF* and *GOTO_IF_NOT*, which perform jumps only when specified conditions are met. These instructions enable the creation of more advanced behaviours, such as complex looping mechanisms. Figure 13 and Figure 14 demonstrate how an if-elif-else test and a for-loop, respectively, can be constructed using a series of labels and *GOTO* instructions.

5 LogicGlue Interpreter

The LogicGlue interpreter runs on the user's microcontroller and translates the instructions in the driver specification into platform-specific commands. The LogicGlue high-level programming library complements the interpreter and offers an interface for developers to interact with electronic components via the interpreter. Unlike high-level communication standards like Jacdac [14], LogicGlue maintains the native signals and features of the components, avoiding the need for translation to a standardized protocol and the associated overhead.

Figure 15 illustrates the LogicGlue program flow for requesting a temperature reading in Kelvin from an I2C temperature sensor that natively reports in Celsius. The process begins with the

LogicGlue Visual Block-based Interface



LogicGlue Driver Specification

```
// Create local variable TMP_0, and global variables VAR_0 and VAR_1
SET_LOCAL(TMP_0), U8(100),
SET_VAR(VAR_0), U8(1),
SET_VAR(VAR_1), U8(10),

// Call function with 1 parameter, where parameter #0 = VAR_0
OP_CALL_ARGS, LABEL_0, 1, VAR_0,

OP_PRINT, LOCAL(TMP_0),      // print(100)   --> local context, variable unchanged
OP_PRINT, VAR(VAR_0),        // print(2)     --> VAR_0 is updated by the function
OP_PRINT, VAR(VAR_1),        // print(10)    --> VAR_1 is unchanged

// Call function with 1 parameter, where parameter #0 = VAR_1
OP_CALL_ARGS, LABEL_0, 1, VAR_1,

OP_PRINT, LOCAL(TMP_0),      // print(100)   --> local context, variable unchanged
OP_PRINT, VAR(VAR_0),        // print(2)     --> VAR_0 is unchanged
OP_PRINT, VAR(VAR_1),        // print(20)    --> VAR_1 is updated by the function
OP_EXIT,

//================= Functions =================//
OP_LABEL, LABEL_0,
    // Create local variable TMP_0, only accessible in this function
    SET_LOCAL(TMP_0), U8(2),

    // Multiply parameter #0 by local variable TMP_0
    SET_PARAM(0),
        MATH_MUL,
            PARAM(0),
            LOCAL(TMP_0),

    OP_RETURN,
```

Fig. 12. Example of the LogicGlue driver specification, demonstrating the scope of variables.

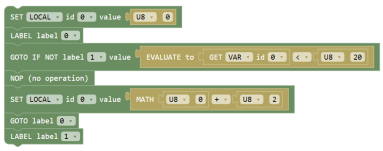
LogicGlue Visual Block-based Interface

LogicGlue Driver Specification

```
// Example of an if-elif-else test that checks if VAR_ is less than 5, 10 or 15.
OP_GOTO_IF_NOT, LABEL_0, EVAL_LT, VAR(VAR_0), U8(5),
    // instructions when VAR_0 is less than 5
    OP_GOTO, LABEL_3,
OP_LABEL, LABEL_0,
    OP_GOTO_IF_NOT, LABEL_1, EVAL_LT, VAR(VAR_0), U8(10),
    // instructions when VAR_0 is less than 10
    OP_GOTO, LABEL_3,
OP_LABEL, LABEL_1,
    OP_GOTO_IF_NOT, LABEL_2, EVAL_LT, VAR(VAR_0), U8(15),
    // instructions when VAR_0 is less than 15
    OP_GOTO, LABEL_3,
OP_LABEL, LABEL_2,
    // instructions for other conditions)
OP_LABEL, LABEL_3,
```

Fig. 13. Example of advanced instructions of the LogicGlue driver specification, demonstrating how an if-elif-else test can be created using *GOTO* instructions and labels.

LogicGlue Visual Block-based Interface



LogicGlue Driver Specification

```
// Example of a for-loop, looping from 0 to 20 with increments of 2
SET_LOCAL(VAR_0), U8(0),
OP_LABEL, LABEL_0,
    OP_GOTO_IF_NOT, LABEL_1, EVAL_LT, VAR(VAR_0), U8(20),
    // instructions inside loop, with VAR_0 the current index
    SET_LOCAL(VAR_0), MATH_ADD, VAR(VAR_0), U8(2),
    OP_GOTO, LABEL_0,
OP_LABEL, LABEL_1,
```

Fig. 14. Example of advanced instructions of the LogicGlue driver specification, demonstrating how a for-loop can be created using *GOTO* instructions and labels.

high-level API receiving a request for the temperature in Kelvin (Step 1). This request is forwarded as a read operation (Step 2) and mapped to the corresponding driver function (Step 3). The driver specification provides the bytecode representation of the function (Step 4), which the interpreter translates into platform-specific commands, including an I2C read to the sensor (Step 5). The sensor

returns the temperature in Celsius (Step 6), prompting the LogicGlue interpreter to apply a data format conversion to Kelvin (Step 7). Finally, the converted value is sent back through the high-level API and returned to the user (Step 8).

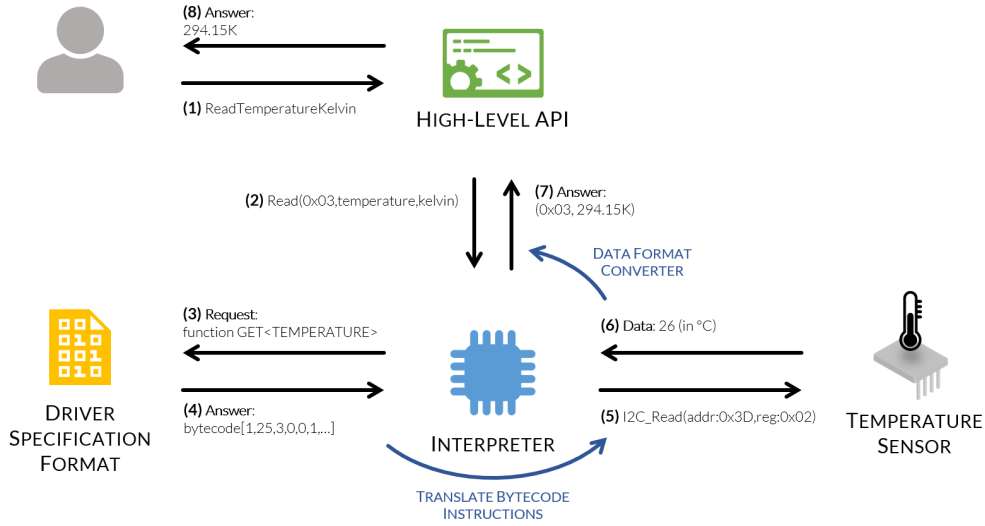


Fig. 15. Illustration of LogicGlue's program flow when requesting a temperature reading in Kelvin from an I2C temperature sensor that natively reports in Celsius.

5.1 LogicGlue High-Level Programming Library

The LogicGlue high-level programming library simplifies interaction with various electronic components by providing functions that serve as wrappers around interpreter calls. These functions handle the initialization, configuration, and operation of components, offering default values for parameters to streamline the process. For instance, Figure 16 illustrates a traditional example of how the driver specification is used in a typical application that changes the colour of an RGB LED based on the measured distance from an ultrasonic distance sensor. This example highlights the standard use of LogicGlue for most components, demonstrating its effectiveness in managing component interactions efficiently.

Initializing components with LogicGlue involves loading the bytecode driver and setting configuration parameters, such as which GPIO pins are connected. This process creates a special environment for the driver to run, managing all the necessary resources, variables, and data buffers for the drivers to work properly. This environment also keeps track of variable states and configurations, ensuring that components operate consistently. This is particularly useful for complex tasks that require multiple steps and need to keep intermediate results. Additionally, the environment can buffer sensor readings, which is helpful for sensors with limited sampling rates, like DHT temperature sensors. In Figure 16, lines 19 and 20 show how the distance sensor and LED are initialized.

After initializing the component, LogicGlue provides specific functions to interact with different types of components. The *create_sensor* function sets up sensors so that data can be read in the right format. The *create_actuator* function sets up actuators, allowing data to be sent to them

```
1  #include <LogicGlue.h>
2  #include <logicglue/drivers/hc-sr04.h>
3  #include <logicglue/drivers/ky-016.h>
4
5  // create the HC-SR04 component connected to pin D8 and D9
6  pinmap_t dist_pinmap = DEFINE_PINMAP(D8, D9);
7  component_t dist_component = DEFINE_COMPONENT(sr04_bytecode, dist_pinmap);
8
9  // create the KY-016 component connected to pin D4, D5 and D6
10 pinmap_t led_pinmap = DEFINE_PINMAP(D4, D5, D6);
11 component_t led_component = DEFINE_COMPONENT(ky016_bytecode, led_pinmap);
12
13 // create a distance sensor and led actuator
14 sensor_t dist_sensor;
15 actuator_t led_actuator;
16
17 void setup() {
18   // initialize the components
19   initialize_component(&dist_component);
20   initialize_component(&led_component);
21
22   // create a new distance sensor for the component
23   create_distance_sensor(&dist_component, &dist_sensor, LENGTH_METRIC, SCALE_CENTI);
24   // create a new led actuator for the component
25   create_led_actuator(&led_component, &led_actuator, COLOR_HSL);
26 }
27
28 void loop() {
29   // get the distance from the sensor and normalize it between 10cm and 110cm
30   float distance = sensor_sample_average(&distance_sensor, 5);
31   distance = min(1.0f, max(0, (distance - 10.0f)) / 100.0f);
32
33   // calculate the hue based on the distance, use color range red (0°) -> blue (240°)
34   uint16_t hue = round(distance * 240.0f);
35   actuator_write_color_hsl(&led_actuator, hue, 100, 20);
36
37   delay(100);
38 }
```

Fig. 16. Example of the application logic for interacting with an ultrasonic distance sensor and RGB LED.

correctly. For displays or other buffer-based components, special functions are available to fill pixel buffers with data. These functions make it easier to work with components by simplifying complex interactions and providing default settings.

In Figure 16, lines 23 and 25 demonstrate how to create a distance sensor that measures in centimetres and an LED actuator that accepts colours in HSL format. Using LogicGlue's high-level functions, interacting with these components becomes straightforward. For example, line 30 shows how the distance sensor is sampled five times to get an average reading. Line 35 shows how the LED colour is set based on the calculated HSL hue value from the distance measurement.

Figure 17 illustrates an example of application logic for interacting with the SSD1306 display using the optimized LogicGlue functions to write a set of colours to the internal display buffers. In this example, a callback function determines the colour of the pixels in the top-left rectangle of the display, while a constant colour is applied to the bottom-right. It is important to note that this exact same code can also be used for other types of displays, such as a dot-matrix display, with no modifications needed beyond the preamble (lines 1-6), as demonstrated in Figure 3. This underscores the flexibility of LogicGlue in managing various components with minimal to no changes to the application logic.

5.2 Converting Data Formats

When using traditional software libraries and drivers to interact with electronic components, developers need to adhere to the data formats outlined by the components. In comparison, LogicGlue allows developers to select their preferred data format for each component in the application logic, independent of the characteristics of the electronic component. For example, in the walkthrough detailed in Section 2, temperature readings are specified in Fahrenheit, as shown in Figure 3, line 15. In contrast, the MCP9808 and TMP36 temperature sensors that are being used, provide readings in Celsius and relative voltages, respectively.

```
#include <LogicGlue.h>
#include <logicglue/drivers/ssd1306.h>

// create the SSD1306 component connected to pin SPI and pin D11, D12 and D13
pinmap_t pinmap = DEFINE_PINMAP(11, 12, 13);
component_t component = DEFINE_COMPONENT(ssd1306_bytecode, pinmap);

display_t display_buffer;
display_t display;

// callback function that determines the color based on the provided coordinates
void calculate_color(uint16_t x, uint16_t y) {
  // all pixels with an odd x-coordinate will be lit
  display_buffer_set_color_binary(&display_buffer, x % 2);
}

void setup() {
  // initialize the component
  initialize_component(&component);

  // create a display buffer and display object for sending the buffer to the display
  create_display_buffer(&component, &display_buffer, COLOR_BINARY);
  create_display(&component, &display);

  // read the width and height of the display
  uint8_t width = display_get_width(&component);
  uint8_t height = display_get_height(&component);


  // clear all pixels in the display
  display_buffer_clear(&display_buffer);

  // draw a rectangle in the top-left corner using a variable color
  display_buffer_fill_rectangle_dynamic(&display_buffer, 0, 0, width/2, height/2, calculate_color);

  // set the color for all next pixel updates (similar to selecting a brush in paint)
  display_buffer_set_color_binary(&display_buffer, 1);
  // draw a rectangle in the bottom-right corner, using the set color
  display_buffer_fill_rectangle(&display_buffer, width/2, height/2, width/2, height/2);

  // write the pixel buffer to the display
  display_update(&display);
}
```

Fig. 17. Example of the application logic for interacting with the SSD1306 OLED display.

To facilitate seamless data conversions, the LogicGlue interpreter automatically applies a series of built-in data converter functions. These functions, all written in bytecode, convert between the data format provided by the application logic and the data format specified in the driver specification. Rather than adopting the impractical approach of including a separate converter function for every possible data format—which would be infeasible given the memory constraints on prototyping platforms—LogicGlue leverages two complementary approaches to reduce the number of required data converters:

(1) **Converting Scalable Formats**:
Scalable data formats use metric prefixes like centi-, milli-, and kilo- to adjust measurement units. Instead of separate converters for each unit pair, LogicGlue uses a generic scale converter that calculates the conversion factor based on these prefixes. This factor is derived from the difference in metric scale steps, each representing a power of 10. For example, converting from milli- to deci- involves moving two steps to the right on the scale, resulting in a conversion factor of $10^2 = 100$. For imperial measurements, LogicGlue uses predefined ratios relative to a base unit (foot). The conversion factor is found by dividing the 'from' ratio by the 'to' ratio. For example, converting inches to miles uses the ratios $\frac{1}{12}$ and 5280, respectively, giving a factor of $\frac{1}{12}/5280 = 0.00001578$. When converting between metric and imperial formats, LogicGlue first converts to the base units before applying the appropriate conversion.

(2) **Converter Chaining**:
LogicGlue automatically chains a set of converters if no single converter is available. For example, if there are converters available for HSL colours to RGB colours and RGB colours

to CMYK colours, LogicGlue automatically chains these to convert HSL colours directly to CMYK colours. Finding a compatible converter chain is managed by representing all data formats as a graph, where each node represents a specific data format, and each edge represents a converter that can transform data from one format to another. To find the most efficient conversion path between two data formats, LogicGlue employs a breadth-first search (BFS) algorithm.

As LogicGlue uses bytecode to represent drivers, the compiler cannot automatically determine which data converters are necessary. Consequently, all available converters are included in the microcontroller's embedded code by default, resulting in significant memory overhead. To optimize memory usage, LogicGlue employs C preprocessor definitions in the driver header files to selectively enable only the relevant converters during the compilation process, ensuring that unnecessary converters are excluded from the final code.

5.3 Interrupt Handling

To enable responsive, event-driven workflows, LogicGlue supports interrupt-driven behaviour for GPIO pins. Interrupts are configured within the LogicGlue driver specification format using the *OP_INTERRUPT* instruction, which defines the pin and the edge condition to monitor. Figure 18 illustrates a complete LogicGlue driver for a button component, including its interrupt configuration for detecting falling edges. At the application level, developers can use the *logicglue_register_interrupt* function to attach a callback that will be executed when the corresponding interrupt is triggered. Figure 19 presents an example of application code that uses this functionality to respond to button presses.

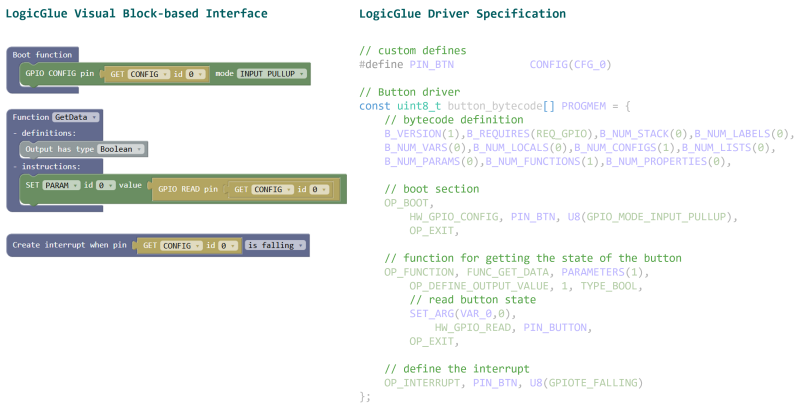


Fig. 18. Logicglue bytecode driver demonstrating how interrupts can be defined.

```
#include <LogicGlue.h>
#include <logicglue/drivers/button.h>

// create the button component connected to pin D4
pinmap_t pinmap = DEFINE_PINMAP(4);
component_t component = DEFINE_COMPONENT(button_bytecode, pinmap);
sensor_t sensor;

// Callback function for button interrupts
void on_button_press() {
  // handle event here...
}

void setup() {
  // initialize the component
  initialize_component(&component);
  // create a new button sensor for the component
  create_button_sensor(&component, &sensor);

  // register the interrupt listener
  logicglue_register_interrupt(pinmap[0], GPIO_EDGE_FALLING, on_button_press);
}

void loop() {
  // ...
}
```

Fig. 19. Example of application code using the LogicGlue bytecode driver for the button, demonstrating how a callback function can be specified for a given interrupt.

6 Supporting LogicGlue on a new Platform

The implementation of the LogicGlue interpreter consists of two parts: (a) An implementation, in C, for parsing the platform-independent driver specifications (bytecodes) and converting data formats.

(b) A platform-specific implementation for communication protocols, GPIO pin access, and memory allocation. This architecture allows for convenient porting of the LogicGlue interpreter to different microcontrollers. Porting LogicGlue to a platform that uses the C programming language only requires implementing the platform-specific functions in Appendix D. Since most microcontroller platforms support C and C++ programming, LogicGlue can be easily implemented on a wide variety of microcontrollers. We already have support for the Arduino and nRF52 platforms.

Supporting LogicGlue is more complex for platforms that run on programming languages that do not build on the C language, such as CircuitPython or DeviceScript. Applications written in these languages are compiled into custom binaries and interpreted on the microcontroller. In these situations, developers must make a one-time effort to fully reimplement both the LogicGlue library and the LogicGlue interpreter in this programming language. Alternatively, LogicGlue could be integrated into the runtime or the language's SDK that executes the custom binaries, as these are typically written in C or C++.

7 LogicGlue Benchmark

While software abstraction layers can introduce some degree of overhead, LogicGlue aims to minimize performance impact by directly interfacing with electronic components using their native protocols and communication signals. Unlike high-level communication protocols such as Jacdac [14], which require translating each interaction into a standardized format, LogicGlue preserves the original communication structure, reducing the need for additional processing. However, unlike traditional C and C++ drivers, which are compiled into optimized machine code that runs directly on the microcontroller, LogicGlue drivers are executed as bytecode by the LogicGlue interpreter at runtime. This means they do not benefit from compiler-level optimizations such as inlining, loop unrolling, and instruction scheduling, resulting in a slight performance overhead compared to low-level drivers. In this section, we benchmark the performance of LogicGlue and demonstrate that its software stack, including the interpreter and high-level programming library, does not significantly impact the performance of interacting with electronic components compared to using component-specific libraries.

To benchmark our system, we measured the execution times for interacting with electronic components using component-specific libraries (baseline condition) and using LogicGlue. We mainly focused on the primary functions of reading and writing data to and from electronic components. Benchmarking initialization procedures is not considered as this typically is a short one-time process and thus has limited impact on the performance. Although LogicGlue embeds many data converters, we did not use this automated conversion of data formats to ensure a fair performance comparison with the baseline condition, as component-specific libraries do not support such features.

We benchmarked the performance of both basic and advanced interactions using three different sensors: an RGB LED controlled via three PWM pins (basic), a DHT22 temperature sensor (intermediate), and an SSD1306 display operating over SPI (advanced). All our tests were conducted on an Arduino Mega microcontroller running the Arduino platform. In the baseline condition, we interact with the RGB LED using Arduino's "AnalogWrite" function. For the DHT22 temperature sensor, we use the Adafruit DHT sensor library[11], and for the SSD1306 display, we use the Adafruit SSD1306 library[12]. For the LogicGlue condition, we used the corresponding LogicGlue bytecode drivers provided in Appendices E, F, and G for the RGB LED, DHT22 temperature sensor and SSD1306 display respectively.

[11] https://github.com/adafruit/DHT-sensor-library
[12] https://github.com/adafruit/Adafruit_SSD1306

For a precise measurement of execution times, we connected a logic analyzer (Saleae Logic Pro 8) to an additional GPIO pin and pulled it to Vcc and ground at, respectively, the start and end of the interaction. The logic analyzer measures the time this GPIO pin is high, which represents the duration of the interaction. This measurement technique is common as it allows for precise and reliable measurements of execution times on a microcontroller. Each test was repeated 100 times for reliability.

Electronic Component	Component-Specific Library	LogicGlue	Difference	Relative Difference
RGB LED	0.03 ms	**0.38 ms**	0.35 ms	+1088%
DHT22 Temperature Sensor	4.26 ms	**4.21 ms**	-0.05 ms	-1.1%
SSD1306 Display	5.44 ms	**8.60 ms**	3.16 ms	+58.1%

Table 1. Execution times for interacting with electronic components using component-specific libraries versus LogicGlue.

The results are summarized in Table 1 and present the median execution times for both the baseline and LogicGlue conditions. Across all components tested, timing variability remained low (±0.002 milliseconds), confirming the consistency and reliability of the measurements. While LogicGlue introduces some overhead, most notably for the SSD1306 display, the overall performance remains within acceptable bounds for typical interactive applications. For instance, the RGB LED operation shows a notable relative increase in execution time due to LogicGlue's additional abstraction layers. However, the absolute time remains under 1 millisecond, which is negligible in most use cases. As a reference point, the frame time of a 60 Hz display is approximately 16 milliseconds, so even a 10× increase here is unlikely to affect perceptual responsiveness in embedded systems.

The DHT22 temperature sensor shows near-identical performance in both configurations, with LogicGlue slightly outperforming the component-specific library, likely due to reduced overhead in its simpler driver structure. In contrast, the SSD1306 display exhibits a measurable increase in latency, from 5.44 ms to 8.60 ms. This reflects the cost of interpreting display-related bytecode and constructing the pixel buffer during execution. Still, the resulting throughput corresponds to a refresh rate above 100 frames per second, far exceeding the practical needs of such low-resolution displays, which are typically updated in response to discrete user actions rather than rendered continuously.

A closer analysis of the results indicates that the primary source of overhead introduced by LogicGlue is not the communication latency itself, but the additional processing required to interpret bytecode instructions and manage component environments. Communication over standard protocols such as I2C, SPI, and GPIO, handled natively by the LogicGlue interpreter, remains largely comparable to component-specific libraries. The physical signalling, bus access time, and device readiness constraints are fundamentally the same in both cases. In contrast, LogicGlue introduces additional logic for evaluating driver bytecode, managing variable environments, and dispatching commands. Even simple operations, such as toggling an LED via PWM, show measurable overhead due to the setup and interpretation stages, despite the actual GPIO or timer interaction being minimal. This illustrates that fixed processing costs, such as initializing execution contexts and interpreting parameter bindings, can dominate performance in lightweight, single-shot operations. In contrast, for more complex drivers, like those managing multi-step I2C transactions or long update

sequences, the relative overhead becomes less significant, as apparent for the DHT22 temperature sensor. The overhead is most apparent in operations involving sequential or looped updates, such as those required to construct and send pixel buffers to displays.

Benchmarking the memory usage of LogicGlue compared to traditional low-level drivers reveals a notable difference in resource consumption. Since LogicGlue drivers are interpreted at runtime, they do not benefit from compiler optimizations such as the zero-overhead abstraction principle in C++. As a result, the LogicGlue software stack requires, dependent on compiler settings and optimizations, approximately 35KB of flash memory when compiled for Arduino and 55KB for the nRF52840 microcontroller. Despite the overhead introduced by the interpreter, LogicGlue bytecode drivers themselves remain compact. The driver for the RGB LED requires only 56 bytes, while the DHT22 temperature sensor and SSD1306 display drivers require 134 bytes and 154 bytes, respectively. These figures highlight the resource efficiency of the driver specifications, even as the interpreter adds some runtime overhead.

8 Discussion and Future Work

8.1 Interchanging Components

LogicGlue's platform-independent drivers allow for writing hardware-independent application logic. This significantly simplifies integrating electronic components, as one does not need to consider technical characteristics, such as protocols and registers. As a result, LogicGlue also facilitates transitions between different microcontrollers and electronic components, which fosters experimentation and iterative development. As such, developers can easily swap components without extensive modifications to the application logic.

Furthermore, because LogicGlue bytecode drivers are processed at runtime, they can be dynamically replaced without the need to recompile or reflash the application logic. This makes it possible to develop applications that update LogicGlue drivers on the fly using, for example, communication protocols like Serial, I2C, or SPI. As a result, testing multiple component types becomes seamless, eliminating the need for manual reprogramming and further enhancing the design of interactive embedded systems.

However, while LogicGlue automatically handles all data format conversions, developers must still understand the functionalities and limitations of both the original and replacement components. For instance, swapping a high-precision I2C temperature sensor with a low-precision analog sensor will result in correct data format conversions. Still, it may impact the application's workings due to the inherent differences in precision.

Display components present another example where careful consideration is needed. Swapping an RGB display with another of a different size requires that the visual interface scales correctly. Replacing an RGB display with a single-colour display necessitates adjustments to accommodate the lack of colour, which LogicGlue can convert but not adapt in terms of design. For example, while LogicGlue automatically converts the RGB colour to a binary colour, it can not adjust the interface to accommodate the lack of colour. Similarly, transitioning from a high-resolution OLED display to a low-resolution dot-matrix display will require application-level modifications to handle the reduced resolution appropriately. Likewise, replacing, for example, a stepper motor with a DC motor involves recognizing the differences in control mechanisms and precision. A stepper motor offers precise position control, which is essential for applications like 3D printing. In contrast, a DC motor provides continuous rotation but lacks the same level of positional accuracy, making it suitable for applications like driving an RC car or a fan.

While LogicGlue enables seamless component interchangeability in most scenarios, some limitations remain. In particular, components that rely on interrupt-driven behaviour cannot be trivially

swapped with components that do not support interrupts. For example, replacing a button driver that uses falling-edge interrupts with a capacitive touch sensor that requires polling would require changes in the application structure. Although the functional goal, detecting user input, may be equivalent, the timing model and event-triggering mechanism differ fundamentally. Future versions of LogicGlue could explore abstractions to unify interrupt- and polling-based components under a shared event model, further enhancing the modularity of interactive systems.

8.2 Performance

LogicGlue introduces a hardware-independent approach that allows application logic to interact with electronic components using their native signals and protocols. As demonstrated in the benchmark (Section 7), its performance closely matches that of component-specific libraries, and for most real-time processing applications, the slight increase in latency is negligible. However, in scenarios requiring strict synchronization, such as certain robotic applications, the minor delay introduced by bytecode interpretation may be a limiting factor, prompting engineers to consider native code for optimal performance. Although our current evaluation focuses on isolated components, we acknowledge the importance of assessing system-wide performance when multiple devices are active. We did not yet benchmark scenarios involving concurrent driver environments or event-driven chains (e.g., a sensor triggers an actuator via an interrupt), but we identify this as a key direction for future work. Particularly, scalability testing under realistic interactive workloads will help further characterize the suitability of LogicGlue in more complex prototyping and deployment scenarios.

Similar challenges arise in alternative approaches like Jacdac [14], which achieves compatibility through an ecosystem of standardized modules. Each module in the Jacdac system contains a dedicated microcontroller that translates component-specific signals into the Jacdac protocol, introducing processing overhead. Additionally, the asynchronous nature of the Jacdac data bus can contribute to further delays and may limit access to component-specific features not explicitly supported by the protocol. Like LogicGlue, Jacdac trades some performance for flexibility, requiring engineers to assess these trade-offs based on their application's timing constraints.

While LogicGlue simplifies hardware interaction, our approach introduces additional memory consumption to store the bytecode, the interpreter, and programming libraries, as discussed in the benchmark (Section 7). Each of these, in turn, introduces additional runtime memory usage. While memory availability on microcontrollers and development boards increases every year, the additional memory consumption of LogicGlue can be an issue for embedded systems with scarce memory resources. To mitigate this issue, future work can look into strategies for optimizing memory usage by refining the bytecode and interpreter. For instance, simplifying the data format converters within LogicGlue could reduce their memory demands. Alternatively, considering the feasibility of offloading certain processing tasks to external devices like a connected computer or leveraging cloud computing resources could help conserve the microcontroller's memory.

8.3 LogicGlue Driver Specification

LogicGlue also eases the work of component manufacturers and engineers of development platforms as electronic components for which LogicGlue driver specifications are written, are instantly compatible with all development platforms that support LogicGlue. Likewise, new development platforms that implement LogicGlue are instantly compatible with the wide variety of electronic components that are on the market today. While LogicGlue requires a one-time effort for a component manufacturer or engineer to write LogicGlue driver specifications for new electronic components, artificial intelligence could be used in the future to generate driver specifications from a component's datasheet automatically.

LogicGlue's driver specifications are stored in bytecode format. They are thus very compact and can be stored in the cloud or on a developer's computer. As these driver specifications include all information to interface with the component, we believe it also makes sense to store this bytecode in the future on a memory chip located on every component. As this would require component manufacturers to follow a new standard, a similar, more practical implementation is the use of a shield that extends any electronic component with a memory chip, as shown in Figure 20. This memory chip could communicate with the LogicGlue Interpreter over I2C or 1-wire, making it possible to recognize plugged-in components and load their bytecode driver specifications automatically. We also envision the memory chip hosting additional information, such as the pinout and operating voltage. This enables new opportunities to assist the wiring process. To further ease or avoid the component wiring, we see opportunities for synergies between our research efforts on lowering the barrier for embedded programming and state-of-the-art solutions to avoid or facilitate component wiring, such as CircuitGlue [24] or VirtualWire [26]. These synergies could lead to novel solutions in which any electronic component becomes plug-and-play, similar to the use of USB to simplify device connectivity.

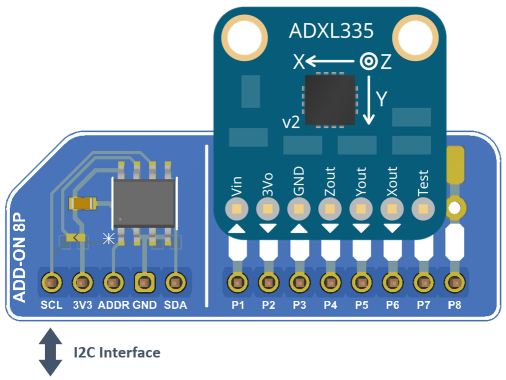


Fig. 20. A conceptual illustration of an extension shield equipped with a memory chip allows for the embedding of a component's driver, ensuring automatic recognition and configuration by LogicGlue upon connection.

9 Conclusion

In this paper we introduced LogicGlue, a novel framework that streamlines electronics prototyping by creating platform-agnostic drivers for hardware components and enabling the development of hardware-independent application logic. At the core of LogicGlue lies a custom driver specification format and interpreter, which enables the definition of a component's functionalities, regardless of platform types or programming languages. Furthermore, we provide a visual block-based programming interface that simplifies writing these specifications. Together, these innovations make the process of developing embedded and interactive systems that draw upon a diverse set of hardware components more accessible to a broad range of individuals.

Acknowledgments

We thank Prof. Dr. Davy Vanacken for providing initial feedback on our paper. This research was supported in part by the Special Research Fund (BOF) project BOF19KP04 and in part by the U.K. Engineering and Physical Sciences Research Council Grant EP/W020564/1.

References

[1] Johnny Accot, Stéphane Chatty, Sébastien Maury, and Philippe Palanque. 1997. Formal Transducers: Models of Devices and Building Bricks for the Design of Highly Interactive Systems. In *Design, Specification and Verification of Interactive Systems '97*, Michael Douglas Harrison and Juan Carlos Torres (Eds.). Springer Vienna, Vienna, 143–159.

[2] Johnny Accot, Stéphane Chatty, and Philippe Palanque. 1996. A Formal Description of Low Level Interaction and its Application to Multimodal Interactive Systems. In *Design, Specification and Verification of Interactive Systems '96*, Francois Bodart and Jean Vanderdonckt (Eds.). Springer Vienna, Vienna, 92–104.

[3] Adafruit. 2024. Adafruit CircuitPython Register. https://github.com/adafruit/Adafruit_CircuitPython_Register

[4] Arduino. 2024. Arduino - Home. https://www.arduino.cc/

[5] Arduino. 2024. Arduino - Libraries. https://www.arduino.cc/reference/en/libraries/

[6] Arduino. 2024. Arduino IDE. https://www.arduino.cc/en/software

[7] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli De Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale. 2020. The BBC micro:bit: from the U.K. to the world. *Commun. ACM* 63, 3 (Feb. 2020), 62–69. doi:10.1145/3368856

[8] Marek Babiuch and Petr Foltynek. 2024. Implementation of a Universal Framework Using Design Patterns for Application Development on Microcontrollers. *Sensors* 24, 10 (2024), 3116. doi:10.3390/s24103116

[9] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michał Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (Athens, Greece) *(SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 7–12. doi:10.1145/3358711.3361630

[10] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 105–116. doi:10.1145/3357390.3361032

[11] BeagleBoard. 2024. BeagleBone. https://www.beagleboard.org/boards/beaglebone-black

[12] Phillip Burgess. 2024. Adafruit GFX Graphics Library. https://learn.adafruit.com/adafruit-gfx-graphics-library/overview

[13] James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. *SIGPLAN Not.* 53, 6 (jun 2018), 19–30. doi:10.1145/3299710.3211335

[14] James Devine, Michal Moskal, Peli de Halleux, Thomas Ball, Steve Hodges, Gabriele D'Amone, David Gakure, Joe Finney, Lorraine Underwood, Kobi Hartley, Paul Kos, and Matt Oppenheim. 2022. Plug-and-play Physical Computing with Jacdac. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 6, 3, Article 110 (9 2022), 30 pages. doi:10.1145/3550317

[15] Pierre Dragicevic and Jean-Daniel Fekete. 2001. Input Device Selection and Interaction Configuration with ICON. In *Proceedings of the International Conference IHM-HCI 2001*, A Blandford, J Vanderdonckt, and P Gray (Eds.). Springer Verlag, Lille, France, 543–448. https://inria.hal.science/hal-00877336

[16] Intel Edison. 2024. Intel Edison Compute Module. https://ark.intel.com/content/www/us/en/ark/products/84572/intel-edison-compute-module-iot.html

[17] FreeRTOS. 2024. FreeRTOS: Real-time operating system for microcontrollers. https://www.freertos.org/index.html

[18] Damien George. 2024. MicroPython: Python for microcontrollers. https://micropython.org/

[19] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 27–36. doi:10.1145/3357390.3361029

[20] Steve Hodges, James Scott, Sue Sentance, Colin Miller, Nicolas Villar, Scarlet Schwiderski-Grosche, Kerry Hammil, and Steven Johnston. 2013. .NET gadgeteer: a new platform for K-12 computer science education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 391–396. doi:10.1145/2445196.2445315

[21] Steve Hodges, Stuart Taylor, Nicolas Villar, James Scott, Dominik Bial, and Patrick Tobias Fischer. 2013. Prototyping Connected Devices for the Internet of Things. *Computer* 46, 2 (2013), 26–34. doi:10.1109/MC.2012.394

[22] Steve Hodges, Nicolas Villar, James Scott, and Albrecht Schmidt. 2012. A New Era for Ubicomp Development. *IEEE Pervasive Computing* 11, 1 (2012), 5–9. doi:10.1109/MPRV.2012.1

[23] Mannu Lambrichts, Raf Ramakers, Steve Hodges, Sven Coppers, and James Devine. 2021. A Survey and Taxonomy of Electronics Toolkits for Interactive and Ubiquitous Device Prototyping. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 2, Article 70 (June 2021), 24 pages. doi:10.1145/3463523

[24] Mannu Lambrichts, Raf Ramakers, Steve Hodges, James Devine, Lorraine Underwood, and Joe Finney. 2023. CircuitGlue: A Software Configurable Converter for Interconnecting Multiple Heterogeneous Electronic Components. *Proc. ACM*

Interact. Mob. Wearable Ubiquitous Technol. 7, 2, Article 63 (6 2023), 30 pages. doi:10.1145/3596265

[25] Mannu Lambrichts, Jose Maria Tijerina, and Raf Ramakers. 2020. SoftMod: A Soft Modular Plug-and-Play Kit for Prototyping Electronic Systems. In *Proceedings of the Fourteenth International Conference on Tangible, Embedded, and Embodied Interaction* (Sydney NSW, Australia) *(TEI '20)*. Association for Computing Machinery, New York, NY, USA, 287–298. doi:10.1145/3374920.3374950

[26] Woojin Lee, Ramkrishna Prasad, Seungwoo Je, Yoonji Kim, Ian Oakley, Daniel Ashbrook, and Andrea Bianchi. 2021. VirtualWire: Supporting Rapid Prototyping with Instant Reconfigurations of Wires in Breadboarded Circuits. In *Proceedings of the Fifteenth International Conference on Tangible, Embedded, and Embodied Interaction* (Salzburg, Austria) *(TEI '21)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. doi:10.1145/3430524.3440623

[27] Philip Levis and David Culler. 2002. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 85–95. doi:10.1145/605397.605407

[28] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. 2005. *TinyOS: An Operating System for Sensor Networks*. Vol. 00. Springer-Verlag, New York, 115–148. doi:10.1007/3-540-27139-2_7

[29] LLVM. 2024. The LLVM Compiler Infrastructure. https://llvm.org/

[30] Elmin Marevac, Esad Kadušić, Nataša Živić, Nevzudin Buzađija, and Samir Lemeš. 2025. Framework Design for the Dynamic Reconfiguration of IoT-Enabled Embedded Systems and "On-the-Fly" Code Execution. *Future Internet* 17, 1 (2025), 1–41. doi:10.3390/fi17010023

[31] Mbed. 2024. Mbed OS. https://os.mbed.com/mbed-os/

[32] BBC micro:bit. 2024. Micro:bit Educational Foundation. https://microbit.org/

[33] Microsoft. 2024. DeviceScript TypeScript for Tiny IoT Devices. https://microsoft.github.io/devicescript/

[34] Microsoft. 2024. Visual Studio Code. https://code.visualstudio.com/

[35] Fumika Mochizuki, Tetsuro Yamazaki, and Shigeru Chiba. 2024. Interactive Programming for Microcontrollers by Offloading Dynamic Incremental Compilation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3679007.3685062

[36] Node-RED. 2024. Node-RED - Low-code programming for event-driven applications. https://nodered.org/

[37] OASIS. 2024. CoAP - Constrained Application Protocol. https://coap.space/

[38] OASIS. 2024. MQTT. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[39] Raspberry Pi. 2024. Raspberry Pi. https://www.raspberrypi.com/

[40] PlatformIO. 2024. PlatformIO. https://platformio.org/

[41] PMod. 2024. PMod. https://digilent.com/reference/pmod/start

[42] Thibault Raffaillac and Stéphane Huot. 2022. What do Researchers Need when Implementing Novel Interaction Techniques? *Proc. ACM Hum.-Comput. Interact.* 6, EICS, Article 159 (June 2022), 30 pages. doi:10.1145/3532209

[43] Jake Robert Read, Leo Mcelroy, Quentin Bolsee, B Smith, and Neil Gershenfeld. 2023. Modular-Things: Plug-and-Play with Virtualized Hardware. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 210, 6 pages. doi:10.1145/3544549.3585642

[44] Steven Varoumas, Basile Pesin, Benoît Vaugon, and Emmanuel Chailloux. 2020. Programming microcontrollers through high-level abstractions. In *Proceedings of the 12th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Virtual, USA) *(VMIL 2020)*. Association for Computing Machinery, New York, NY, USA, 5–14. doi:10.1145/3427765.3428495

[45] Zephyr. 2024. Device Driver Model. https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/drivers/index.html

[46] Zephyr. 2024. Zephyr. https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/index.html

A Instructions in the Driver Specification

```
OP_NOP()
  No operation
OP_BOOT()
  Marks the start of the boot section
OP_FUNCTION(name:8, args:8, props:8)
  Define a function with arguments and properties
OP_INTERRUPT(pin:num, event:num)
  Define an interrupt for the provided pin and event
OP_PROPERTY(name:8, n:8, accepts:8[n])
  Define a property for the current function
OP_PROPERTY_CONST(name:8, val:num)
  Define a constant property for the current function
OP_DEFINE_INPUT_VALUE(num:8, (type:8)[n])
  Define the type of value of the input parameters
OP_DEFINE_INPUT_FORMAT(num:8, (format:8, scale:8)[n])
  Define the format and scale of input parameters
OP_DEFINE_OUTPUT_VALUE(num:8, (type:8)[n])
  Define the type of value of the output parameters
OP_DEFINE_OUTPUT_FORMAT(num:8, (format:8, scale:8)[n])
  Define the format and scale of output parameters
OP_DEFINE_FUNCTION_TYPE(type:8)
  Define the type of the function
OP_SET_VAR(id:8, val:num)
  Save value in a variable
OP_SET_VAR_INITIAL(id:8, val:num)
  Save value only if the variable is not set
OP_SET_ARG(id:8, idx:8, val:num)
  Save value in argument at specified index
OP_SET_PROP(val:num)
  Save value in the current property
OP_SET_LOCAL(id:8, val:num)
  Save value in a local variable
OP_SET_PARAM(id:8, val:num)
  Save value in a cross-context parameter
OP_SET_LIST(id:8, lst:list)
  Save list in a list variable
OP_LIST_CREATE_1D(id:8, type:8(list_e), n:num)
  Create a 1D list
OP_LIST_CREATE_2D(id:8, type:8(list_e), n:num, m:num)
  Create a 2D list
OP_LIST_SET_1D(id:8, idx:num, val:num)
  Set value in a 1D list
OP_LIST_SET_2D(id:8, idx:num, idy:num, val:num)
  Set value in a 2D list
OP_LIST_FILL(id:8, val:num)
  Fill list with value
```

```
OP_PRINT(val:num)
  Print the value
OP_PRINT_LIST(lst:list)
  Print the list
OP_LABEL(label:8)
  Mark a label
OP_GOTO(label:8)
  Go to a label
OP_GOTO_IF(label:8, val:num)
  Conditional goto if truthy
OP_GOTO_IF_NOT(label:8, val:num)
  Conditional goto if falsy
OP_CALL(label:8)
  Call label and store index on stack
OP_CALL_ARGS(label:8, n:8, idx[n])
  Call label with argument list
OP_CALL_INTERNAL(internal:8)
  Call internal function
OP_CALL_INTERNAL_ARGS(internal:8, n:8, idx[n])
  Call internal with arguments
OP_RETURN()
  Return from current call
OP_BLOCK(n:8, instruction[n])
  Execute a block of instructions
OP_BREAK()
  Exit the current block or return
OP_BREAK_IF(val:num)
  Break if truthy
OP_IF(val:num, instruction)
  Execute instruction if truthy
OP_IF_ELSE(val:num, instruction, instruction)
  If-else conditional
OP_IF_ELIF_ELSE(n:8, (val:num, instruction)[n], instruction)
  If-elif-else with default
OP_SWITCH(n:8, switch:num, (case:num, instruction)[n], instruction)
  Switch-case logic
OP_LOOP(start:num, end:num, incr:num, instruction)
  Loop over range
OP_FOREACH(lst:list, instruction)
  Loop over list items
OP_FOREACH_BYTE(lst:list, instruction)
  Loop over list bytes
OP_EXIT()
  Exit the program
OP_EXIT_CODE(code:num)
  Exit with code
OP_ASSERT(val:num)
  Assert value is truthy
```

```
OP_ASSERT_CODE(val:num, code:num)
  Assert value or exit with code
HW_DELAY_MS(duration:num)
  Delay in milliseconds
HW_DELAY_US(duration:num)
  Delay in microseconds
HW_GPIO_CONFIG(pin:num, mode:num)
  Configure GPIO pin mode
HW_GPIO_WRITE(pin:num, state:num)
  Set GPIO pin state
HW_GPIO_PULSE_MS(pin:num, state:num, duration:num)
  Pulse pin for milliseconds
HW_GPIO_PULSE_MS_n(pin:num, state:num, duration:num, interval:num, n:num)
  Repeated pulse in ms
HW_GPIO_PULSE_US(pin:num, state:num, duration:num)
  Pulse pin for microseconds
HW_GPIO_PULSE_US_n(pin:num, state:num, duration:num, interval:num, n:num)
  Repeated pulse in µs
HW_GPIOTE_CONFIG(pin:num, edge:num, label:8)
  Configure edge-triggered GPIO interrupt
HW_ADC_CONFIG(pin:num)
  Configure ADC on pin
HW_PWM_CONFIG(pin:num, period:num)
  Configure PWM signal
HW_PWM_WRITE(pin:num, duty:num)
  Set PWM duty cycle
HW_I2C_CONFIG(addr:num, freq:8)
  Configure I2C interface
HW_I2C_WRITE(val:num)
  Write single value to I2C
HW_I2C_WRITE_2(val:num, val:num)
  Write two values to I2C
HW_I2C_WRITE_LIST(lst:list)
  Write list to I2C
HW_SPI_CONFIG(cs:num, freq:8, mode:8, order:8)
  Configure SPI interface
HW_SPI_WRITE(val:num)
  Write value to SPI
HW_SPI_WRITE_2(val:num, val:num)
  Write two values to SPI
HW_SPI_WRITE_LIST(lst:list)
  Write list to SPI
HW_UART_CONFIG(baudrate:8, mode:8)
  Configure UART
HW_UART_WRITE(val:num)
  Write to UART
HW_UART_WRITE_2(val:num, val:num)
  Write two values to UART
```

```
HW_UART_WRITE_LIST(lst:list)
  Write list to UART
HW_DTH_CONFIG(pin:num)
  Configure DTH sensor
HW_WS2812_CONFIG(pin:num)
  Configure WS2812 LED
HW_WS2812_WRITE(val:num)
  Write value to WS2812
HW_WS2812_WRITE_LIST(lst:list)
  Write list to WS2812
```

Listing 1. Full instruction set with parameters and descriptions

B Numeric Subsystem in the Driver Specification

```
Instruction(\textbf{Parameters})
 Description
_U8(val:8)
 Define an 8-bit unsigned integer constant
_U16(val:16)
 Define a 16-bit unsigned integer constant
_U32(val:32)
 Define a 32-bit unsigned integer constant
_I8(val:8)
 Define an 8-bit signed integer constant
_I16(val:16)
 Define a 16-bit signed integer constant
_I32(val:32)
 Define a 32-bit signed integer constant
_FLT(val:32)
 Define a 32-bit floating point constant
_FLT_HEX(val:32)
 Define a 32-bit floating point constant in hexadecimal format
_FIX(val:32, frac:8)
 Define a fixed-point constant
_CONFIG(id:8)
 Get configuration value
_ARG(id:8, idx:num)
 Get the value at specified index from the argument
_PROP(No parameters)
 Get the value of the current property
_VAR(id:8)
 Get variable value
_VAR_LOCAL(id:8)
 Get local variable value
_VAR_PARAM(id:8)
 Get parameter value
LIST_GET_1D(id:8, idx:num)
 Get the value at specified index from the 1D list
LIST_GET_2D(id:8, idx:num, idy:num)
 Get the value at specified index from the 2D list
LIST_SIZE(id:8)
 Get the number of items in the list
CAST(type:8(num_e), val:num)
 Cast the numeric value to the given type
MATH_ADD(val_a:num, val_b:num)
 Add two numeric values
MATH_SUB(val_a:num, val_b:num)
 Subtract second numeric value from the first
MATH_MUL(val_a:num, val_b:num)
 Multiply two numeric values
```

```
MATH_DIV(val_a:num, val_b:num)
 Divide the first numeric value by the second
MATH_MOD(val_a:num, val_b:num)
 Get the remainder of division of the first numeric value by the second
MATH_POW(val_a:num, val_b:num)
 Raise the first numeric value to the power of the second
MATH_SQRT(val:num)
 Get the square root of the numeric value
MATH_AND(val_a:num, val_b:num)
 Perform bitwise AND operation on two numeric values
MATH_OR(val_a:num, val_b:num)
 Perform bitwise OR operation on two numeric values
MATH_XOR(val_a:num, val_b:num)
 Perform bitwise XOR operation on two numeric values
MATH_NOT(val:num)
 Perform bitwise NOT operation on the numeric value
MATH_SHL(val_a:num, val_b:num)
 Shift the first numeric value left by the number of bits specified by the second
MATH_SHR(val_a:num, val_b:num)
 Shift the first numeric value right by the number of bits specified by the second
MATH_MIN(val_a:num, val_b:num)
 Get the minimum of two numeric values
MATH_MAX(val_a:num, val_b:num)
 Get the maximum of two numeric values
MATH_CLAMP(val:num, low:num, high:num)
 Clamp the numeric value within the range specified by low and high values
MATH_ABS(val:num)
 Get the absolute value of the numeric value
MATH_CEIL(val:num)
 Get the ceiling value of the numeric value
MATH_FLOOR(val:num)
 Get the floor value of the numeric value
MATH_ROUND(val:num)
 Round the numeric value
MATH_SIN(val:num)
 Get the sine of the numeric value
MATH_COS(val:num)
 Get the cosine of the numeric value
MATH_TAN(val:num)
 Get the tangent of the numeric value
MATH_ASIN(val:num)
 Get the arcsine of the numeric value
MATH_ACOS(val:num)
 Get the arccosine of the numeric value
MATH_ATAN(val:num)
 Get the arctangent of the numeric value
MATH_ATAN2(val:num)
 Get the arctangent of the quotient of the two numeric values
```

```
BITS_BIT(val:num, n:num)
 Get the value of the nth bit of the numeric value
BITS_MASK_NEG(type:8(num_e), mask:num)
 Create a mask with background 1 and the nth bit 0
BITS_MASK_POS(type:8(num_e), mask:num)
 Create a mask with background 0 and the nth bit 1
IF_EQ(val_a:num, val_b:num, val_if:num, val_else:num)
 Evaluate to val_if if val_a == val_b, otherwise evaluate to val_else
IF_NE(val_a:num, val_b:num, val_if:num, val_else:num)
 Evaluate to val_if if val_a != val_b, otherwise evaluate to val_else
IF_GT(val_a:num, val_b:num, val_if:num, val_else:num)
 Evaluate to val_if if val_a > val_b, otherwise evaluate to val_else
IF_GE(val_a:num, val_b:num, val_if:num, val_else:num)
 Evaluate to val_if if val_a >= val_b, otherwise evaluate to val_else
IF_LT(val_a:num, val_b:num, val_if:num, val_else:num)
 Evaluate to val_if if val_a < val_b, otherwise evaluate to val_else
IF_LE(val_a:num, val_b:num, val_if:num, val_else:num)
 Evaluate to val_if if val_a <= val_b, otherwise evaluate to val_else
BOOL_AND(val_a:num, val_b:num)
 Evaluate to the boolean val_a AND val_b
BOOL_OR(val_a:num, val_b:num)
 Evaluate to the boolean val_a OR val_b
BOOL_NOT(val:num)
 Evaluate to the boolean NOT val
EVAL_EQ(val_a:num, val_b:num)
 Evaluate to 1 if val_a == val_b, otherwise 0
EVAL_NE(val_a:num, val_b:num)
 Evaluate to 1 if val_a != val_b, otherwise 0
EVAL_GT(val_a:num, val_b:num)
 Evaluate to 1 if val_a > val_b, otherwise 0
EVAL_GE(val_a:num, val_b:num)
 Evaluate to 1 if val_a >= val_b, otherwise 0
EVAL_LT(val_a:num, val_b:num)
 Evaluate to 1 if val_a < val_b, otherwise 0
EVAL_LE(val_a:num, val_b:num)
 Evaluate to 1 if val_a <= val_b, otherwise 0
NUM_SWITCH(n:8, switch:num, (case:num, val:num)[n], def:num)
 Switch the value over cases and evaluate to val if switch == case, if no match,
    evaluate to def
NUM_LOOP_IDX_0()
 Get the current loop index 0
NUM_LOOP_IDX_1()
 Get the current loop index 1
HW_MILLIS()
 Returns the number of milliseconds since the board was powered up
HW_MICROS()
 Returns the number of microseconds since the board was powered up
HW_GPIO_READ(pin:num)
```

```
 Read the value of a GPIO pin
HW_GPIO_PULSE_READ(pin:num, state:num)
 Get the duration of a pulse on the pin
HW_GPIO_PULSE_READ_T(pin:num, state:num, timeout:num)
 Get the duration of a pulse on the pin or timeout
HW_ADC_READ(pin:num)
 Read the value of an ADC pin
HW_I2C_READ_U8()
 Read an 8-bit unsigned integer from the I2C device
HW_I2C_READ_U16()
 Read a 16-bit unsigned integer from the I2C device
HW_I2C_READ_U32()
 Read a 32-bit unsigned integer from the I2C device
HW_I2C_READ_I8()
 Read an 8-bit signed integer from the I2C device
HW_I2C_READ_I16()
 Read a 16-bit signed integer from the I2C device
HW_I2C_READ_I32()
 Read a 32-bit signed integer from the I2C device
HW_I2C_READ_FLT()
 Read a 32-bit floating point number from the I2C device
HW_I2C_WRITE_READ_U8(val:num)
 Write a numeric to the I2C device, then read an 8-bit unsigned integer
HW_I2C_WRITE_READ_U16(val:num)
 Write a numeric to the I2C device, then read a 16-bit unsigned integer
HW_I2C_WRITE_READ_U32(val:num)
 Write a numeric to the I2C device, then read a 32-bit unsigned integer
HW_I2C_WRITE_READ_I8(val:num)
 Write a numeric to the I2C device, then read an 8-bit signed integer
HW_I2C_WRITE_READ_I16(val:num)
 Write a numeric to the I2C device, then read a 16-bit signed integer
HW_I2C_WRITE_READ_I32(val:num)
 Write a numeric to the I2C device, then read a 32-bit signed integer
HW_I2C_WRITE_READ_FLT(val:num)
 Write a numeric to the I2C device, then read a 32-bit floating point number
HW_SPI_READ_U8()
 Read an 8-bit unsigned integer from the SPI device
HW_SPI_READ_U16()
 Read a 16-bit unsigned integer from the SPI device
HW_SPI_READ_U32()
 Read a 32-bit unsigned integer from the SPI device
HW_SPI_READ_I8()
 Read an 8-bit signed integer from the SPI device
HW_SPI_READ_I16()
 Read a 16-bit signed integer from the SPI device
HW_SPI_READ_I32()
 Read a 32-bit signed integer from the SPI device
HW_SPI_READ_FLT()
```

```
 Read a 32-bit floating point number from the SPI device
HW_SPI_WRITE_READ_U8(val:num)
 Write a numeric to the SPI device, then read an 8-bit unsigned integer
HW_SPI_WRITE_READ_U16(val:num)
 Write a numeric to the SPI device, then read a 16-bit unsigned integer
HW_SPI_WRITE_READ_U32(val:num)
 Write a numeric to the SPI device, then read a 32-bit unsigned integer
HW_SPI_WRITE_READ_I8(val:num)
 Write a numeric to the SPI device, then read an 8-bit signed integer
HW_SPI_WRITE_READ_I16(val:num)
 Write a numeric to the SPI device, then read a 16-bit signed integer
HW_SPI_WRITE_READ_I32(val:num)
 Write a numeric to the SPI device, then read a 32-bit signed integer
HW_SPI_WRITE_READ_FLT(val:num)
 Write a numeric to the SPI device, then read a 32-bit floating point number
HW_UART_READ_U8()
 Read an 8-bit unsigned integer from the UART device
HW_UART_READ_U16()
 Read a 16-bit unsigned integer from the UART device
HW_UART_READ_U32()
 Read a 32-bit unsigned integer from the UART device
HW_UART_READ_I8()
 Read an 8-bit signed integer from the UART device
HW_UART_READ_I16()
 Read a 16-bit signed integer from the UART device
HW_UART_READ_I32()
 Read a 32-bit signed integer from the UART device
HW_UART_READ_FLT()
 Read a 32-bit floating point number from the UART device
```

Listing 2. Numeric instructions with parameters and descriptions

C List Subsystem in the Driver Specification

```
_LIST_U8(n:8)
 Create a list of 8-bit unsigned integers
_LIST_U16(n:8)
 Create a list of 16-bit unsigned integers
_LIST_U32(n:8)
 Create a list of 32-bit unsigned integers
_LIST_I8(n:8)
 Create a list of 8-bit signed integers
_LIST_I16(n:8)
 Create a list of 16-bit signed integers
_LIST_I32(n:8)
 Create a list of 32-bit signed integers
_LIST_FLT(n:8)
 Create a list of 32-bit floating point numbers
_LIST(id:8)
 Get list with specified id
_LIST_PARAM(id:8)
 Get list parameter with specified id
HW_I2C_READ_LIST()
 Read a list from the I2C device
HW_I2C_WRITE_READ_LIST()
 Write a list to the I2C device, then read a list
HW_SPI_READ_LIST()
 Read a list from the SPI device
HW_SPI_WRITE_READ_LIST()
 Write a list to the SPI device, then read a list
HW_UART_READ_LIST()
 Read a list from the UART device
HW_UART_WRITE_READ_LIST()
 Write a list to the UART device, then read a list
HW_DTH_READ_LIST(variant:8, pin:num)
 Read a list from the DHT sensor with specified variant and pin
```

Listing 3. List instructions with parameters and descriptions

D Platform-Specific Functions

```
//=== GPIO ===//
status_e target_gpio_config(const uint8_t pin, const gpio_mode_e mode);
status_e target_gpio_write(const uint8_t pin, const uint8_t state);
status_e target_gpio_read(const uint8_t pin, uint8_t* state);
status_e target_gpio_read_pulse(const uint8_t pin, const uint8_t state, uint32_t* pulse);
status_e target_gpio_read_pulse_timeout(const uint8_t pin, const uint8_t state, const uint32_t timeout, uint32_t* pulse);
//=== GPIOTE ===//
status_e target_gpiote_config(const uint8_t pin, const gpio_edge_e edge, const gpio_irq_f callback);
//=== PWM ===//
status_e target_pwm_config(const uint8_t pin, const uint16_t period);
status_e target_pwm_write(const uint8_t pin, const uint16_t duty);
//=== ADC ===//
status_e target_adc_config(const uint8_t pin);
status_e target_adc_read(const uint8_t pin, uint16_t* value);
//=== I2C ===//
status_e target_i2c_config(const i2c_freq_e freq, const uint8_t mtu);
status_e target_i2c_write(const uint8_t address, uint8_t* buffer, uint16_t len);
status_e target_i2c_read(const uint8_t address, uint8_t* buffer, uint16_t len);
status_e target_i2c_write_read(const uint8_t address, uint8_t* write_buffer, uint16_t write_len, uint8_t* read_buffer, uint16_t read_len);
//=== SPI ===//
status_e target_spi_config(const spi_freq_e freq, const spi_mode_e mode, const spi_order_e order, const uint8_t mtu);
status_e target_spi_write(const uint8_t cs, uint8_t* buffer, uint16_t len);
status_e target_spi_read(const uint8_t cs, uint8_t* buffer, uint16_t len);
status_e target_spi_write_read(const uint8_t cs, uint8_t* write_buffer, uint16_t write_len, uint8_t* read_buffer, uint16_t read_len);
//=== UART ===//
status_e target_uart_config(const uart_baud_e baud, const uart_mode_e mode, const uint8_t mtu);
status_e target_uart_write(const uint8_t id, uint8_t* buffer, uint16_t len);
status_e target_uart_read(const uint8_t id, uint8_t* buffer, uint16_t len);
status_e target_uart_write_read(const uint8_t id, uint8_t* write_buffer, uint16_t write_len, uint8_t* read_buffer, uint16_t read_len);
//=== DHT ===//
status_e target_dht_config(const uint8_t pin);
status_e target_dht_read(const dth_variant_e variant, const uint8_t pin, uint8_t* buffer);

//=== TIME ===//
void target_sleep_ms(uint32_t ms);
void target_sleep_us(uint32_t us);
uint32_t target_millis();
uint64_t target_micros();
//=== MEMORY ===//
void* target_malloc(uint16_t size);
void* target_calloc(uint16_t num, uint16_t size);
void* target_realloc(void* ptr, uint16_t size);
void target_free(void* ptr);
int target_get_free_memory(void);
uint16_t target_get_memory_counter(void);
//=== INTERRUPTS ===//
void target_disable_interrupts(void);
void target_enable_interrupts(void);
uint8_t target_in_interrupt(void);
```

Fig. 21. Header file detailing the platform-specific functions needed to be implemented when porting LogicGlue to a new platform.

E KY-016 RGB LED LogicGlue Driver

```
#include "../common.h"
#include "../language.h"

// custom defines
#define PIN_RED       CONFIG(CFG_0)
#define PIN_GREEN     CONFIG(CFG_1)
#define PIN_BLUE      CONFIG(CFG_2)

// KY-016 RGB LED module driver
static const uint8_t ky016_bytecode[] PROGMEM = {
  // bytecode size in bytes
  U16_ARR(56),

  // bytecode definition
  B_VERSION(1),           // version
  B_REQUIRES(REQ_PWM),        // hardware requirements
  B_NUM_STACK(0),           // number of stack elements
  B_NUM_LABELS(0),          // number of labels
  B_NUM_VARS(0),            // number of variables
  B_NUM_LOCALS(0),          // number of local variables
  B_NUM_CONFIGS(3),         // number of config variables
  B_NUM_LISTS(0),           // number of lists
  B_NUM_PARAMS(0),          // number of parameters
  B_NUM_FUNCTIONS(1),         // number of functions
  B_NUM_PROPERTIES(0),        // number of properties

  // boot section
  OP_BOOT,
    HW_PWM_CONFIG, PIN_RED, U8(255),
    HW_PWM_CONFIG, PIN_GREEN, U8(255),
    HW_PWM_CONFIG, PIN_BLUE, U8(255),
    OP_EXIT,

  // function for setting the color in RGB format
  OP_FUNCTION, FUNC_SET_DATA, PARAMETERS(1),
    OP_DEFINE_INPUT_FORMAT, 1, COLOR_RGB, SCALE_NONE,

    HW_PWM_WRITE, PIN_RED, ARG(ARG_0,0),
    HW_PWM_WRITE, PIN_GREEN, ARG(ARG_0,1),
    HW_PWM_WRITE, PIN_BLUE, ARG(ARG_0,2),
    OP_EXIT,OP_EXIT,
};
```

Fig. 22. LogicGlue driver for the KY-016 RGB LED.

F DHT22 Temperature Sensor LogicGlue Driver

```
#include "../common.h"
#include "../language.h"

// custom defines
#define DHT22_PIN     CONFIG(CFG_0)
#define DHT22_READ      LABEL_0

// DHT22 temperature and humidity sensor driver
static const uint8_t dht22_bytecode[] PROGMEM = {
  // bytecode size in bytes
  U16_ARR(134),

  // bytecode definition
  B_VERSION(1),           // version
  B_REQUIRES(REQ_DHT),        // hardware requirements
  B_NUM_STACK(1),           // number of stack elements
  B_NUM_LABELS(1),          // number of labels
  B_NUM_VARS(1),            // number of variables
  B_NUM_LOCALS(1),          // number of local variables
  B_NUM_CONFIGS(1),         // number of config variables
  B_NUM_LISTS(1),           // number of lists
  B_NUM_PARAMS(0),          // number of parameters
  B_NUM_FUNCTIONS(2),         // number of functions
  B_NUM_PROPERTIES(0),        // number of properties

  // boot section
  OP_BOOT,
    HW_DTH_CONFIG, DHT22_PIN,
    OP_EXIT,

  // Read the temperature from DHT22 sensor
  OP_FUNCTION, FUNC_GET_DATA, PARAMETERS(1),
    OP_DEFINE_OUTPUT_FORMAT, 1, TEMPERATURE_CELSIUS, SCALE_UNIT,

    OP_CALL, DHT22_READ,
    SET_ARG(ARG_0,0),
      MATH_MUL,
        MATH_MUL,
          MATH_OR,
            MATH_SHL, CAST,TYPE_U16,
              MATH_AND,
                LIST_GET_1D,LIST_0, U8(2),
                U8(0x7f),
              U8(8),
            LIST_GET_1D,LIST_0, U8(3),
          FLT(0.1f),
        IF_EQ, MATH_AND, LIST_GET_1D,LIST_0, U8(2), U8(0x80), U8(0),
          U8(1),
          I8(-1),
    OP_EXIT,

  // Read the humidity from DHT22 sensor
  OP_FUNCTION, FUNC_GET_DATA, PARAMETERS(1),
    OP_DEFINE_OUTPUT_FORMAT, 1, HUMIDITY_PERCENT, SCALE_UNIT,
```

Fig. 23. LogicGlue driver for the DHT22 Temperature Sensor (part 1/2).

```
    OP_CALL, DHT22_READ,
    SET_ARG(ARG_0,0),
      MATH_MUL,
        MATH_OR,
          MATH_SHL, CAST,TYPE_U16,
            LIST_GET_1D,LIST_0, U8(0),
            U8(8),
          LIST_GET_1D,LIST_0, U8(1),
        FLT(0.1f),
    OP_EXIT,

  // Read data from DHT22 sensor
  OP_LABEL, DHT22_READ,
    SET_VAR_INITIAL(VAR_0), U8(0),

    COMMAND("data can only be read once every 2 seconds")
    // if (millis() - last_read < 2000 || VAR_0 == 0) { ... }
    OP_IF,
      BOOL_OR,
        EVAL_GT, MATH_SUB, HW_MILLIS, VAR(VAR_0), U16(2000),
        EVAL_EQ, VAR(VAR_0), U8(0),
      OP_BLOCK, 2,
        SET_LIST(LIST_0), HW_DTH_READ_LIST,DHT_22, DHT22_PIN,
        SET_VAR(VAR_0), HW_MILLIS,

    OP_RETURN,
};
```

Fig. 24. LogicGlue driver for the DHT22 Temperature Sensor (part 2/2).

G SSD1306 Display LogicGlue Driver

```
#include "../common.h"
#include "../language.h"

// custom defines
#define SSD1306_DC_PIN        CONFIG(CFG_0)
#define SSD1306_CS_PIN        CONFIG(CFG_1)
#define SSD1306_RST_PIN       CONFIG(CFG_2)
#define SSD1306_WRITE         LABEL_0

// SSD1306 OLED display driver
static const uint8_t ssd1306_bytecode[] PROGMEM = {
  // bytecode size in bytes
  U16_ARR(154),

  // bytecode definition
  B_VERSION(1),           // version
  B_REQUIRES(REQ_GPIO,REQ_SPI),   // hardware requirements
  B_NUM_STACK(1),           // number of stack elements
  B_NUM_LABELS(3),          // number of labels
  B_NUM_VARS(0),            // number of variables
  B_NUM_LOCALS(1),          // number of local variables
  B_NUM_CONFIGS(3),         // number of config variables
  B_NUM_LISTS(1),           // number of lists
  B_NUM_PARAMS(0),          // number of parameters
  B_NUM_FUNCTIONS(2),         // number of functions
  B_NUM_PROPERTIES(2),        // number of properties

  // boot section
  OP_BOOT,
    HW_SPI_CONFIG, SSD1306_CS_PIN, U8(SPI_FREQ_8M), U8(SPI_MODE_0), U8(SPI_ORDER_MSB),
U8(0),
    HW_GPIO_CONFIG, SSD1306_DC_PIN, U8(GPIO_MODE_OUTPUT),
    HW_GPIO_CONFIG, SSD1306_CS_PIN, U8(GPIO_MODE_OUTPUT),
    HW_GPIO_CONFIG, SSD1306_RST_PIN, U8(GPIO_MODE_OUTPUT),
    // reset display
    HW_GPIO_WRITE, SSD1306_RST_PIN, U8(PIN_HIGH),
    HW_DELAY_MS, U8(1),
    HW_GPIO_WRITE, SSD1306_RST_PIN, U8(PIN_LOW),
    HW_DELAY_MS, U8(10),
    HW_GPIO_WRITE, SSD1306_RST_PIN, U8(PIN_HIGH),
    // enable command mode
    HW_GPIO_WRITE, SSD1306_DC_PIN, U8(PIN_LOW),
    HW_SPI_WRITE_LIST, LIST_U8(26),
      0xAE,   // display off
      0xD5,   // set display clock div
      0x80,   // the suggested ratio 0x80
      0xA8,   // set multiplex
      0x3F,   // 63
      0xD3,   // set display offset
      0x00,   // no offset
      0x40,   // set start line
      0x8D,   // charge pump
      0x14,   // enable charge pump
      0x20,   // memory mode
```

Fig. 25. LogicGlue driver for the SSD1306 Display (part 1/2).

```
      0x00,   // horizontal addressing
      0xA1,   // set segment remap
      0xC8,   // set com output scan direction
      0xDA,   // set com pins
      0x12,   // com pins hardware configuration
      0x81,   // set contrast
      0xCF,   // contrast level
      0xD9,   // set pre-charge
      0xF1,   // pre-charge level
      0xDB,   // set vcom detect
      0x40,   // vcom detect
      0xA4,   // display all on resume
      0xA6,   // normal display
      0x2E,   // deactivate scroll
      0xAF,   // display on
    OP_LIST_CREATE_2D, LIST_0, LIST_TYPE_U1_Y, U8(128), U8(64),
    OP_CALL, SSD1306_WRITE, // transmit data buffer
    OP_EXIT,

  COMMENT("Update display")
  OP_FUNCTION, FUNC_UPDATE_DATA, PARAMETERS(0),
    OP_CALL, SSD1306_WRITE,
    OP_EXIT,

  COMMENT("Set pixel color in data buffer")
  OP_FUNCTION, FUNC_SET_DATA, PARAMETERS(1),
    OP_DEFINE_INPUT_FORMAT, 1, COLOR_BINARY, SCALE_NONE,
    OP_DEFINE_FUNCTION_TYPE, FUNC_TYPE_LIST_2D,
    OP_EXIT,

  COMMENT("Transmit data buffer to display")
  OP_LABEL, SSD1306_WRITE,
    // enable command mode, prepare data transfer
    HW_GPIO_WRITE, SSD1306_DC_PIN, U8(PIN_LOW),
    HW_SPI_WRITE_LIST, LIST_U8(6),
      0x21,   // set column address
      0,    // start at 0
      127,  // end at 127
      0x22,   // set page address
      0,    // start at 0
      7,    // end at 7
    // enable data mode, transfer data
    HW_GPIO_WRITE, SSD1306_DC_PIN, U8(PIN_HIGH),
    HW_SPI_WRITE_LIST, LIST(LIST_0),
    OP_EXIT,

  COMMENT("property for getting the width of the SSD1306 display")
  OP_PROPERTY_CONST, PROP_GET_WIDTH, U8(128),
  COMMENT("property for getting the height of the SSD1306 display")
  OP_PROPERTY_CONST, PROP_GET_HEIGHT, U8(64),
};
```

Fig. 26. LogicGlue driver for the SSD1306 Display (part 2/2).