

# SRAM-Pico2W

## Technical Documentation

**Target Platform:** Raspberry Pi Pico 2 / Pico 2 W (RP2350)  
**Architecture:** ARM Cortex-M33  
**Author:** rewalo

SRAM-Pico2W is a firmware architecture for the Raspberry Pi Pico 2 that enables running Arduino-style applications entirely from Static RAM (SRAM) while maintaining a minimal, flash-resident kernel. This design provides a clean separation between a persistent kernel layer and an application layer, enabling rapid development cycles and efficient memory utilization.

This documentation is maintained alongside the source code.

# Contents

<b>1</b>	<b>Abstract</b>	<b>4</b>
<b>2</b>	<b>User's Manual</b>	<b>4</b>
2.1	Prerequisites	4
2.2	Building an Application	4
2.2.1	Step 1: Compile Application	4
2.2.2	Step 2: Upload Kernel	5
2.2.3	Step 3: Monitor Output	5
2.3	Editing Application Code	5
2.4	Adding New Syscalls	5
<b>3</b>	<b>System Architecture</b>	<b>5</b>
3.1	Memory Layout	6
3.1.1	Flash Memory Organization	6
3.1.2	SRAM Memory Organization	6
3.2	Runtime Execution Flow	6
3.3	Component Interaction	7
<b>4</b>	<b>Build System</b>	<b>8</b>
4.1	Build Pipeline	8
4.2	Critical Address Alignment	8
4.3	Build Configuration	9
<b>5</b>	<b>Syscall System</b>	<b>9</b>
5.1	Syscall Definition Format	9
5.2	Code Generation Process	9
5.2.1	1. Enum Generation (syscall_ids.h)	10
5.2.2	2. App-Side Wrappers (app_syscalls.h)	10
5.2.3	3. Low-Level Gate (app_sys_raw.c)	10
5.2.4	4. Kernel Dispatch (kernel_sys_jumtable.inc.h)	11
<b>6</b>	<b>Type Marshalling</b>	<b>11</b>
<b>7</b>	<b>Application Binary Format</b>	<b>12</b>
7.1	Header Structure	12
7.2	Section Layout	13
<b>8</b>	<b>Proxy System for C++ Objects</b>	<b>13</b>
8.1	Design Pattern	13
<b>9</b>	<b>FreeRTOS Integration</b>	<b>14</b>
9.1	Task Structure	14
<b>10</b>	<b>Memory Safety and Security</b>	<b>14</b>
10.1	Current Safety Mechanisms	14
10.1.1	Header Validation	14
10.1.2	Syscall Isolation	14
10.1.3	Pointer Validation	14
10.2	Security Considerations	15
<b>11</b>	<b>Build Configuration</b>	<b>15</b>
11.1	Compiler Flags	15
11.2	Linker Configuration	16
<b>12</b>	<b>Debugging and Diagnostics</b>	<b>16</b>
12.1	Syscall Tracing	16
12.2	Common Failure Modes	16

<b>13 Performance Characteristics</b>	<b>17</b>
13.1 Memory Usage	17
13.2 Syscall Overhead	17
13.3 Further Optimization Opportunities	17
<b>14 Future Enhancements</b>	<b>17</b>
14.1 Planned Features	17
14.2 API Expansion	18
<b>15 Code Style and Standards</b>	<b>18</b>
<b>16 Supported Platforms</b>	<b>18</b>
16.1 Target Hardware	18
16.2 Development Environment	18
<b>17 References</b>	<b>18</b>
<b>18 License</b>	<b>19</b>

# 1 Abstract

---

SRAM-Pico2W is a two-layer firmware architecture for the Raspberry Pi Pico 2 and Pico 2 W (RP2350 microcontroller). Its primary purpose is to enable running Arduino-style applications entirely from Static RAM (SRAM) while maintaining all Arduino core libraries and hardware drivers in flash memory. This design provides a clean separation between a persistent kernel layer and an application layer that can be updated by rebuilding and reflashing the kernel.

The system uses a syscall-based communication mechanism that allows applications to request hardware services from the kernel. Applications are compiled separately and embedded into the kernel binary at build time, then loaded into SRAM at runtime. This architecture enables rapid development cycles where only the application code needs to be rebuilt, while the kernel remains stable in flash memory.

For deployment, a single build script compiles the application and embeds it into the kernel, requiring only the kernel binary to be uploaded to the device flash memory.

## 2 User's Manual

---

This chapter provides step-by-step instructions for building, deploying, and using SRAM-Pico2W applications.

### 2.1 Prerequisites

- Windows computer (build scripts use batch files)
- Python 3 installed
- Arduino IDE installed
- RP2040 board support installed in Arduino IDE:
  - Board package: **Raspberry Pi Pico/RP2040** by **Earle Philhower**
- Raspberry Pi Pico 2 or Pico 2 W
- USB cable (data-capable, not charge-only)

### 2.2 Building an Application

#### 2.2.1 Step 1: Compile Application

Execute `build.bat` from the project root directory:

```
1 | build.bat
```

This script performs the following operations:

1. Generates syscall wrapper code from `syscalls.def`
2. Compiles the application code (`app/app.ino`) and support files
3. Links the application with a custom SRAM memory map
4. Converts the ELF binary to raw binary format
5. Embeds the binary into `kernel/src/generated/rawData.h`

The build process shows progress for each step and reports the final binary size.

### 2.2.2 Step 2: Upload Kernel

1. Open `kernel/kernel.ino` in Arduino IDE
2. Select board: Tools > Board > Raspberry Pi RP2040 Boards > Raspberry Pi Pico 2W
3. Select COM port: Tools > Port > [your Pico's port]
4. Click Upload (Ctrl+U)

The kernel includes the embedded application binary and loads it into SRAM at boot.

### 2.2.3 Step 3: Monitor Output

1. Open Serial Monitor: Tools > Serial Monitor (Ctrl+Shift+M)
2. Set baud rate to 115200
3. Press Reset button on Pico
4. Observe boot messages and application output

## 2.3 Editing Application Code

Application code is located in `app/app.ino`. After editing:

1. Run `build.bat` to rebuild the application
2. Upload the kernel again (application is embedded in kernel)
3. Reset the device to load the new application

## 2.4 Adding New Syscalls

To expose additional Arduino functions to applications:

1. Open `syscalls.def`
2. Add a new line: `SYSCALL(name, returnType, (args...))`
3. Run `build.bat`
4. Upload kernel
5. Use the function in application code

### Syscall Examples

```
1 SYSCALL(analogRead, int, (int pin))
2 SYSCALL(analogWrite, void, (int pin, int value))
```

### Syscall Constraints

- Maximum 30 arguments per syscall
- Arguments must be primitive types or pointers
- Return types must fit in `uintptr_t` or be `void`

## 3 System Architecture

---

SRAM-Pico2W implements a two-layer architecture that separates persistent kernel services from application code:

- **Kernel Layer** (Flash): Persistent, contains Arduino core, hardware drivers, and embedded application binary
- **Application Layer** (SRAM): Loaded at runtime, contains only application code and data

## 3.1 Memory Layout

### 3.1.1 Flash Memory Organization

Flash memory contains:

- Bootstrap code (Arduino runtime initialization)
- FreeRTOS kernel
- Application loader (`loader_rawdata.cpp`)
- Syscall dispatcher (`kernel_syscall_dispatch.cpp`)
- Arduino Core libraries (Serial, GPIO, timing, etc.)
- Embedded application binary (`rawData[]` array)

### 3.1.2 SRAM Memory Organization

SRAM is organized starting at address `0x20030000`:

```
0x20030000: +-----+
             | .app_hdr      | Application header + syscall gate
             +-----+
             | .text         | Application code
             +-----+
             | .data         | Initialized variables
             +-----+
             | .bss          | Zero-initialized data
             +-----+
             | __heap_start__ | Dynamic allocation (malloc/free)
             | ...           |
             | __heap_end__   | End of SRAM (0x200A0000)
             +-----+
```

#### Memory Statistics

**Total SRAM Size:** 448 KB (0x00070000 bytes)

## 3.2 Runtime Execution Flow

1. **Boot:** RP2350 ROM initializes hardware and loads kernel from flash
2. **Kernel setup():**
  - Initializes serial communication (115200 baud)
  - Copies embedded application binary from flash (`rawData[]`) to SRAM at `0x20030000`
    - Binary copy includes `.text` (code), `.data` (initialized variables), and `.app_hdr` (header)
  - Zeros the BSS section (uninitialized static variables) using boundaries from app header
  - Validates application header (magic number `0x41505041` = “APPA”)
  - Validates version number (`0x00020000`)
  - Patches syscall gate pointer into application header

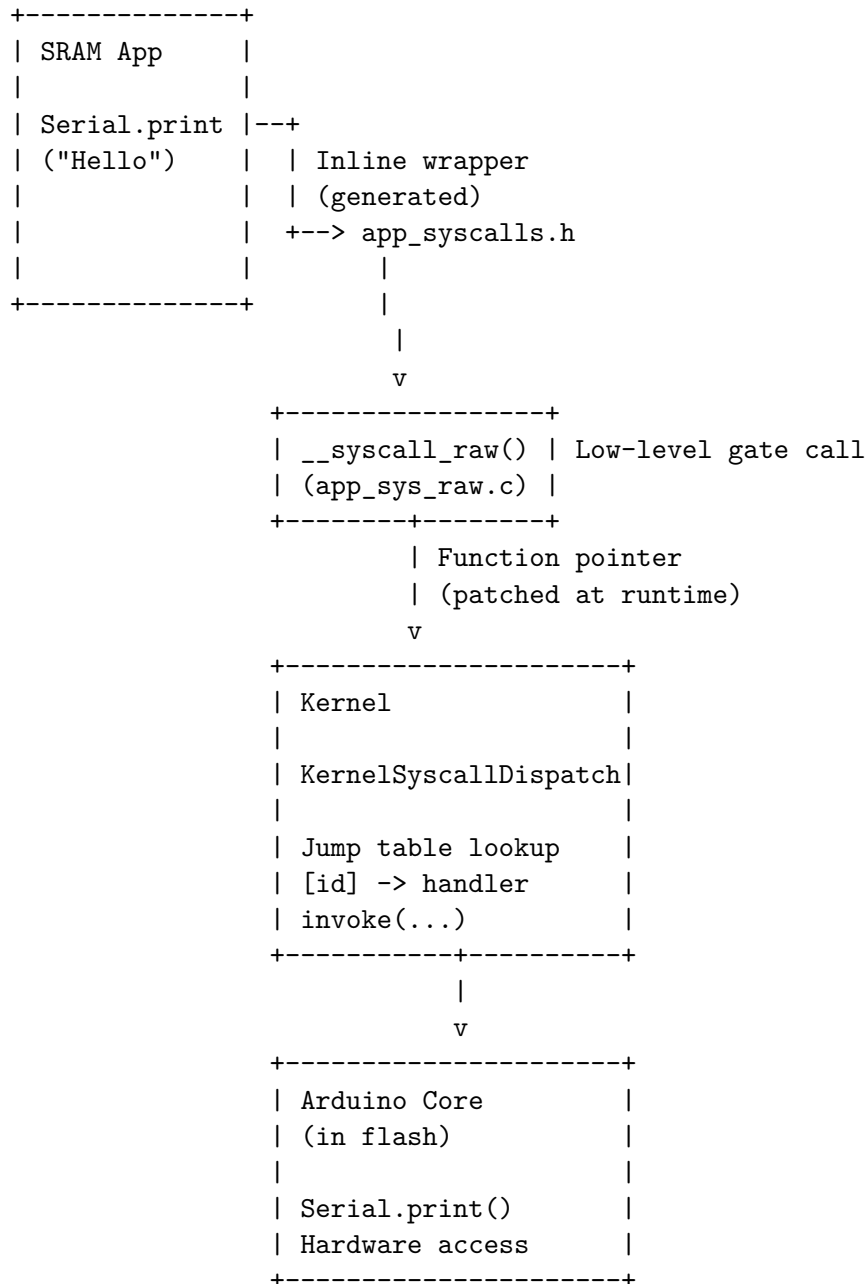
- Calls `app_setup()` once in kernel context
- Launches FreeRTOS task for `app_loop()`

### 3. Application Execution:

- Application code executes entirely from SRAM
- Hardware access via syscall gate → kernel dispatcher
- Kernel services execute from flash-resident code

## 3.3 Component Interaction

The syscall mechanism works as follows:



## 4 Build System

---

## 4.1 Build Pipeline



## 4.2 Critical Address Alignment

All components must agree on the SRAM base address 0x20030000:

Component	File	Location
Linker Script	app/linker/memmap_app_ram.ld	ORIGIN = 0x20030000
Loader	kernel/src/loader_rawdata.cpp	kAppDst = 0x20030000
Kernel	kernel/kernel.ino	kAppBaseAddr = 0x20030000



## Address Validation

The kernel performs the following validation checks:

- Header magic validation: `0x41505041` = “APPA”
- Version validation: `0x00020000`
- Mismatched addresses cause “Invalid app header” error

## 4.3 Build Configuration

The build system supports the following configuration options in `build.bat`:

- **LIBC**: Enable/disable libc linking (ON/OFF)
- **OPT\_LEVEL**: Optimization level (Os = size, O2 = speed, O0 = debug)
- **CUSTOM\_LIBS**: Additional library paths
- **CUSTOM\_INCLUDES**: Additional include directories
- **VERBOSE**: Verbose build output (ON/OFF)

## 5 Syscall System

---

The syscall system provides a controlled interface between SRAM-resident applications and flash-resident kernel services. This architecture design:

1. Prevents direct hardware access from application code
2. Enables centralized hardware management in the kernel
3. Allows kernel updates without recompiling applications (with versioning)
4. Minimizes SRAM footprint by keeping Arduino core in flash

### 5.1 Syscall Definition Format

Syscalls are defined in `syscalls.def` using the following format:

```
1 | SYSCALL(name, return_type, (arg1_type arg1, arg2_type arg2, ...))
```

### Syscall Constraints

- Maximum 30 arguments (system limit defined by `MAX_SYSCALL_ARGS`)
- Arguments must be primitive types or pointers
- Return types must fit in `uintptr_t` or be `void`

### Syscall Examples

```
1 | SYSCALL(pinMode,          void,      (uint8_t pin, uint8_t mode))
2 | SYSCALL(digitalWrite,    void,      (uint8_t pin, uint8_t val))
3 | SYSCALL(digitalRead,     int,        (uint8_t pin))
4 | SYSCALL(Serial_print_s,  size_t,    (const char* s))
```

### 5.2 Code Generation Process

### 5.2.1 1. Enum Generation (syscall\_ids.h)

The generator creates an enum with unique IDs for each syscall:

```
1 enum : unsigned {
2     SYSC_pinMode = 0,
3     SYSC_digitalWrite = 1,
4     SYSC_digitalRead = 2,
5     ...
6     SYSC__COUNT = 10
7 };
```

#### Critical Constraint

Enum order must match `syscalls.def` order. Changing the order in `syscalls.def` changes syscall IDs and breaks binary compatibility between applications and kernel.

### 5.2.2 2. App-Side Wrappers (app\_syscalls.h)

The generator creates inline wrapper functions for applications:

```
1 static inline void pinMode(uint8_t pin, uint8_t mode) {
2     uintptr_t _p0 = static_cast<uintptr_t>(pin);
3     uintptr_t _p1 = static_cast<uintptr_t>(mode);
4     uintptr_t _p2 = 0;
5     uintptr_t _p3 = 0;
6     // ... (all 30 parameters: _p0 through _p29, unused ones set to 0)
7     uintptr_t _p29 = 0;
8     __syscall_raw(SYSC_pinMode, _p0, _p1, _p2, _p3, _p4, _p5,
9                   _p6, _p7, _p8, _p9, _p10, _p11, _p12, _p13, _p14,
10                  _p15,
11                  _p16, _p17, _p18, _p19, _p20, _p21, _p22, _p23, _p24,
12                  _p25,
13                  _p26, _p27, _p28, _p29);
14 }
```

#### Type Safety

- Pointers: `reinterpret_cast<uintptr_t>(ptr)`
- Primitives: `static_cast<uintptr_t>(value)`

### 5.2.3 3. Low-Level Gate (app\_sys\_raw.c)

The gate function calls the syscall dispatcher via function pointer. The gate pointer is cached on first call to avoid repeated lookups:

```
1 static Gate cached_gate = NULL;
2
3 uintptr_t __syscall_raw(uint16_t id, uintptr_t a0, uintptr_t a1,
4                         uintptr_t a2, uintptr_t a3, uintptr_t a4,
5                         uintptr_t a5, uintptr_t a6, uintptr_t a7,
6                         // ... (30 arguments total: a0 through a29)
7                         uintptr_t a28, uintptr_t a29) {
8     if (cached_gate == NULL) {
9         cached_gate = __syscall_gate_ptr();
10    }
11 }
```

```

11     return cached_gate(id, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
12                        a10, a11, a12, a13, a14, a15, a16, a17, a18, a19
13                        ,
14                        a20, a21, a22, a23, a24, a25, a26, a27, a28, a29
15                        );
16 }

```

#### 5.2.4 4. Kernel Dispatch (kernel\_sys\_jumtable.inc.h)

The kernel dispatcher uses a jump table for direct function pointer dispatch, avoiding switch statement overhead:

```

1  // Handler function for each syscall (takes all 30 arguments: a0
2  //   through a29)
3  static inline uintptr_t handlePinMode(uintptr_t a0, uintptr_t a1,
4  //                                     uintptr_t a2, ..., uintptr_t
5  //                                     a29) {
6  //     return detail::invoke<&::pinMode>(a0, a1, a2, a3, a4, a5, ..., a29)
7  //     ;
8  // }
9
10 // Jump table array - direct indexing by syscall ID
11 static const SyscallHandler syscallHandlers[] = {
12     &handlePinMode,
13     &handleDigitalWrite,
14     // ...
15 };
16
17 // Dispatch function uses direct array lookup
18 if (id >= SYSC_COUNT) return -1;
19 return syscallHandlers[id](a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
20                          a10, a11, a12, a13, a14, a15, a16, a17,
21                          a18, a19,
22                          a20, a21, a22, a23, a24, a25, a26, a27,
23                          a28, a29);

```

## 6 Type Marshalling

---

Arguments are passed as `uintptr_t` values and unmarshalled by the kernel using template metaprogramming:

```
1 template <typename T>
2 static inline T cast_arg_uintptr(uintptr_t v) {
3     using U = std::remove_reference_t<T>;
4     if constexpr (std::is_pointer_v<U>) {
5         return reinterpret_cast<T>(v); // Pointers
6     } else if constexpr (std::is_enum_v<U>) {
7         return static_cast<T>(v);      // Enums
8     } else if constexpr (std::is_integral_v<U>) {
9         return static_cast<T>(v);      // Integers
10    } else {
11        static_assert(!sizeof(U), "Unsupported syscall arg type");
12    }
13 }
```

### Type Marshalling Limitations

- Complex structs by value are not supported (would require serialization)
- Arrays must be passed as pointers
- Callbacks/function pointers require special handling

## 7 Application Binary Format

### 7.1 Header Structure

The application header is located at the start of the SRAM image in the `.app_hdr` section:

```
1 typedef struct AppHeader {
2     uint32_t magic;           // 'APPA' = 0x41505041
3     uint32_t version;        // 0x00020000 (ABI version)
4     void (*app_setup)(void);
5     void (*app_loop)(void);
6     SyscallGate syscall_gate; // Patched by kernel at runtime
7     void* data_start;         // Start of .data section
8     void* data_end;           // End of .data section
9     void* bss_start;          // Start of BSS section (for zeroing)
10    void* bss_end;             // End of BSS section (for zeroing)
11    uint32_t* bss_marker;      // Pointer to BSS verification marker
12 } AppHeader;
```

### Header Layout

- **Magic number:** Authentication check (must be 0x41505041)
- **Version:** ABI compatibility validation (must be 0x00020000)
- **Function pointers:** Entry points for kernel invocation (`app_setup`, `app_loop`)
- **Syscall gate:** Runtime-patched function pointer to kernel dispatcher
- **Section boundaries:** Pointers to `.data` and `.bss` section boundaries for initialization
- **BSS marker:** Pointer to verification marker used to confirm BSS zeroing

## 7.2 Section Layout

```
0x20030000: +-----+
             | .app_hdr      | Header + syscall gate
             +-----+
             | .text         | Application code
             +-----+
             | .data         | Initialized variables
             +-----+
             | .bss          | Zero-initialized data
             +-----+
             | __heap_start__| Dynamic allocation
             | ...           |
             | __heap_end__  | End of SRAM
             +-----+
```

## 8 Proxy System for C++ Objects

---

### 8.1 Design Pattern

Arduino-style C++ objects (e.g., `Serial`) are implemented using lightweight proxy classes that forward calls to syscalls:

```
1 struct SerialProxy {
2     inline void begin(unsigned long baud) {
3         Serial_begin(baud);
4     }
5     inline size_t print(const char* s) {
6         return Serial_print_s(s);
7     }
8     inline void println(const char* s) {
9         return Serial_println_s(s);
10    }
11    // ...
12 };
13
14 inline SerialProxy Serial; // Global instance
```

Usage in Application:

```
1 Serial.begin(115200); // Calls syscall
2 Serial.println("Hello"); // Calls syscall
```

#### Proxy System Benefits

- Natural API usage (no change to application code style)
- Zero runtime overhead (inline functions)
- Type safety maintained

## 9 FreeRTOS Integration

---

### 9.1 Task Structure

The kernel uses FreeRTOS to manage application execution:

```
1 static void appTask(void*) {
2     AppHeader* hdr = reinterpret_cast<AppHeader*>(kAppBaseAddr);
3     for (;;) {
4         if (hdr->app_loop != nullptr) {
5             hdr->app_loop();
6         }
7         vTaskDelay(pdMS_TO_TICKS(1)); // Cooperative yield
8     }
9 }
```

#### FreeRTOS Design Decisions

- Application loop() runs in separate FreeRTOS task
- 1ms cooperative yield prevents task starvation
- Kernel loop() provides watchdog/management functions

## 10 Memory Safety and Security

---

### 10.1 Current Safety Mechanisms

#### 10.1.1 Header Validation

##### Header Validation

- Magic number prevents executing invalid data
- Version check prevents ABI mismatches
- Invalid header causes kernel to enter error state (LED blink)

#### 10.1.2 Syscall Isolation

##### Syscall Isolation

- Applications cannot directly access hardware
- All hardware operations go through kernel
- Kernel validates pointer ranges for string operations

#### 10.1.3 Pointer Validation

The kernel validates application-provided pointers before dereferencing:

```
1 inline bool isValidAppPointer(const void* ptr, size_t len) {
2     if (ptr == nullptr) return false;
```

```

3  uintptr_t addr = reinterpret_cast<uintptr_t>(ptr);
4  if (addr < kSramBaseAddr || addr >= kSramEndAddr) return false;
5  if (len > 0 && addr > kSramEndAddr - len) return false;
6  return true;
7  }

```

### Pointer Validation Scope

This validation is applied to syscalls that accept string/buffer pointers (e.g., `Serial_print_s`).

## 10.2 Security Considerations

### Current Security Limitations

- No bounds checking on all buffer pointers
- No validation of pointer ranges for all syscalls
- Syscall IDs not cryptographically signed
- No privilege separation beyond syscall boundary

### Recommended Security Enhancements

- Add bounds checking for all buffer operations
- Validate pointer ranges against known safe regions for all syscalls
- Implement syscall rate limiting
- Add application signing/verification

## 11 Build Configuration

### 11.1 Compiler Flags

Application code is compiled with the following flags:

```

CFLAGS:
-mcpu=cortex-m33           # RP2350 CPU architecture
-mthumb                   # Thumb instruction set
-mfloat-abi=softfp        # Software floating point
-DNO_STDIO                 # Disable stdio functions
-Os -g0                   # Optimize for size, no debug
-fdata-sections           # Enable section garbage collection
-ffunction-sections       # Enable function section GC
-fno-exceptions           # Disable C++ exceptions
-fno-rtti                 # Disable runtime type info

```

## Libc Support

When libc is enabled, the application can use standard C library functions with custom implementations provided in `newlib_stubs.c`.

## 11.2 Linker Configuration

```
LD_FLAGS:
  -nostdlib           # No standard library
  -nostartfiles       # Custom startup code
  -Wl,--gc-sections   # Remove unused sections
  -T"memmap_app_ram.ld" # Custom memory map
```

## Optional Linker Flags

- `-lc -lgcc`: Link against libc (requires `newlib_stubs.c`)

# 12 Debugging and Diagnostics

## 12.1 Syscall Tracing

Enable debug output in `kernel/src/kernel_syscall_dispatch.cpp`:

```
1 #define DEBUG_SYSCALLS
```

### Output Format:

```
[SYSC] 3 (pinMode) args=[13,1,0,0,0,0]
[SYSC] ret=0
```

## 12.2 Common Failure Modes

Symptom	Likely Cause	Solution
“Invalid app header”	Address mismatch	Verify 0x20030000 in all locations
“Unknown syscall”	ID mismatch	Regenerate syscall files
App crashes	Invalid pointer passed	Add bounds checking
Undefined symbols	Missing generated files	Run <code>build.bat</code>
Upload fails	Wrong board selected	Select Pico 2W in Arduino IDE



## 13 Performance Characteristics

---

### 13.1 Memory Usage

#### Memory Footprint

- **Kernel:** 50–100 KB flash (varies with Arduino core features)
- **Application:** Typically 10–50 KB SRAM (code + data)
- **Available Heap:** 400 KB (448 KB total - app footprint)

### 13.2 Syscall Overhead

#### Performance Metrics

- **Gate lookup:** 2–5 cycles (cached after first call, 1 cycle subsequently)
- **Function call:** 50–100 CPU cycles
- **Dispatch:** 3–5 cycles (direct jump table lookup + type casting)
- **Type marshalling:** 5–15 cycles (template-based argument conversion)
- **Total latency:** 60–125 cycles ( $\approx 0.5\text{--}1.0\ \mu\text{s}$  @ 120 MHz)

#### Optimizations Applied

The following optimizations reduce syscall overhead:

- **Gate pointer caching:** Eliminates repeated header dereferences
- **Jump table dispatch:** Direct array indexing (predictable branches)

### 13.3 Further Optimization Opportunities

1. Variable argument passing (only pass needed arguments instead of all 30)
2. Fast-path inlining for high-frequency syscalls (bypass template machinery)
3. Direct dispatch code generation (eliminate template overhead for common cases)
4. Register calling convention optimization (ensure optimal register usage)

## 14 Future Enhancements

---

### 14.1 Planned Features

1. **Hot Reload:** Upload apps without reflashing kernel
2. **Multiple App Slots:** Runtime switching between applications
3. **Syscall Versioning:** Automatic ABI compatibility checking
4. **Secure Boot:** Cryptographic verification of applications
5. **Protected Memory:** Hardware-based memory protection

## 14.2 API Expansion

- WiFi/Network stack syscalls
- File system operations
- Peripheral drivers (SPI, I2C, etc.)
- Interrupt handling
- Power management

## 15 Code Style and Standards

---

### Code Style Standards

This project adheres to the **Google C++ Style Guide**:

- 2-space indentation
- Pointer style: `char* ptr` (not `char *ptr`)
- C++ casts: `static_cast`, `reinterpret_cast`
- Include order: system headers, then project headers
- Function names: camelCase
- Constants: `kConstantName`

All generated code follows these conventions automatically.

## 16 Supported Platforms

---

### 16.1 Target Hardware

- Raspberry Pi Pico 2 (RP2350)
- Raspberry Pi Pico 2 W (RP2350 with WiFi)

### 16.2 Development Environment

- Windows (build scripts)
- Arduino IDE 1.8.x or 2.x
- Python 3.x
- RP2040 board support package (Earle Philhower)

## 17 References

---

- [Raspberry Pi Pico 2 Documentation](#)
- [Arduino RP2040 Core](#)
- [Google C++ Style Guide](#)
- [FreeRTOS Documentation](#)

## 18 License

---

SRAM-Pico2W is licensed under the GNU General Public License v3.0.

### GNU General Public License v3.0

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

The full license text is available at <https://www.gnu.org/licenses/gpl-3.0.html>.