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

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:

```
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 (includes code overlay region)
4. Converts the ELF binary to raw binary format
5. Generates page map metadata (`page_gen.py`) for code paging system
6. 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

```
SYSCALL(analogRead, int, (int pin))
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
            +------------------+
            | .data            | Initialized variables (fixed address)
            +------------------+
            | .bss             | Zero-initialized data (fixed address)
            +------------------+
            | __heap_start__   | Dynamic allocation (malloc/free)
            | ...              |
            | ...              |
            +------------------+
0x20040000: | CODE OVERLAY     | Code execution window (256 KB)
            | WINDOW           | All code pages load here
            |                  | Pages overlay each other
            +------------------+
            | ...              |
            | __heap_end__     | End of SRAM (0x200A0000)
            +------------------+
```

Memory Statistics

- **Total SRAM Size:** 448 KB (0x00070000 bytes)
- **Code Overlay Window:** 256 KB at `0x20040000`
- **Header/Data Region:** Fixed addresses starting at `0x20030000`

3.2 Runtime Execution Flow

1. **Boot:** RP2350 ROM initializes hardware and loads kernel from flash
2. **Kernel setup():**
   - Initializes serial communication (115200 baud)

- Initializes code overlay system (`code_cache_init()`)
- Copies header and data sections from flash (`rawData[]`) to fixed SRAM addresses
  - Header section (`.app_hdr`) copied to `0x20030000`
  - Data section (`.data`) copied to fixed addresses after header
  - Code pages are **not** copied yet - loaded on demand via overlay
- Zeros the BSS section (uninitialized static variables) using boundaries from app header
- Loads critical code pages (containing `app_setup()` and `app_loop()`) into overlay window
- 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 from SRAM overlay window (`0x20040000`)
   - Code pages are loaded on demand when functions are called
   - Overlay system manages page loading and eviction transparently
   - Hardware access via syscall gate → kernel dispatcher
   - Kernel services execute from flash-resident code

3.3 Component Interaction

The syscall mechanism works as follows:

```
+--------------+
| SRAM App     |
|              |
| Serial.print |--+
| ("Hello")    |  | Inline wrapper
|              |  | (generated)
|              |  +--> app_syscalls.h
|              |        |
+--------------+        |
                         |
                         v
                 +-----------------+
                 | __syscall_raw() | Low-level gate call
                 | (app_sys_raw.c) |
                 +--------+--------+
                          | Function pointer
                          | (patched at runtime)
                          v
                 +----------------------+
                 | Kernel               |
                 |                      |
                 | KernelSyscallDispatch|
                 |                      |
                 | Jump table lookup    |
                 | [id] -> handler      |
                 | invoke(...)          |
                 +-----------+----------+
                             |
                             v
                 +-----------------------+
                 | Arduino Core          |
                 | (in flash)            |
                 |                       |
                 | Serial.print()        |
                 | Hardware access       |
                 +-----------------------+
```

4 Build System

4.1 Build Pipeline

```
+--------------+
| syscalls.def| Master definition file
+------+-------+
       |
       v
+-------------------+
| syscall_gen.py    | Python generator
+------+------------+
       |
       +--> app/src/generated/app_syscalls.h
       +--> app/src/generated/app_sys_raw.c
       +--> app/src/generated/syscall_ids.h
       +--> kernel/src/generated/kernel_sys_dispatch.inc.h
       +--> kernel/src/generated/syscall_ids.h
       +--> kernel/src/generated/syscall_names.inc.h
       |
       v
+-------------------+
| build.bat         | Build orchestration
+------+------------+
       |
       +--> Compiles app.ino + support files
       +--> Links with custom linker script (SRAM + overlay)
       +--> Generates app.elf
       +--> Converts to app.bin
       +--> Generates page map (page_gen.py)
       |     --> kernel/src/generated/app_page_map.h
       |     --> kernel/src/generated/app_page_map.c
       +--> Embeds app.bin -> kernel/src/generated/rawData.h
       |
       v
+-------------------+
| Arduino IDE       | Kernel compilation
+------+------------+
       |
       +--> Compiles kernel.ino + all kernel sources
       +--> Includes embedded rawData.h
```

```
    +--> Uploads to device flash
```

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:

```
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

```
SYSCALL(pinMode,        void,    (uint8_t pin, uint8_t mode))
SYSCALL(digitalWrite,   void,    (uint8_t pin, uint8_t val))
SYSCALL(digitalRead,    int,     (uint8_t pin))
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:

```
enum : unsigned {
  SYSC_pinMode = 0,
  SYSC_digitalWrite = 1,
  SYSC_digitalRead = 2,
  ...
  SYSC__COUNT = 10
};
```

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:

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

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:

```c
static Gate cached_gate = NULL;

uintptr_t __syscall_raw(uint16_t id, uintptr_t a0, uintptr_t a1,
                        uintptr_t a2, uintptr_t a3, uintptr_t a4,
                        uintptr_t a5, uintptr_t a6, uintptr_t a7,
                        // ... (30 arguments total: a0 through a29)
                        uintptr_t a28, uintptr_t a29) {
  if (cached_gate == NULL) {
    cached_gate = __syscall_gate_ptr();
  }
  return cached_gate(id, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
                     a10, a11, a12, a13, a14, a15, a16, a17, a18, a19,
                     a20, a21, a22, a23, a24, a25, a26, a27, a28, a29);
}
```

5.2.4 4. Kernel Dispatch (`kernel_sys_jumptable.inc.h`)

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

```cpp
// Handler function for each syscall (takes all 30 arguments: a0 through a29)
static inline uintptr_t handlePinMode(uintptr_t a0, uintptr_t a1,
                                       uintptr_t a2, ..., uintptr_t a29) {
  return detail::invoke<&::pinMode>(a0, a1, a2, a3, a4, a5, ..., a29);
}

// Jump table array - direct indexing by syscall ID
static const SyscallHandler syscallHandlers[] = {
  &handlePinMode,
  &handleDigitalWrite,
  // ...
};

// Dispatch function uses direct array lookup
if (id >= SYSC__COUNT) return -1;
return syscallHandlers[id](a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
                           a10, a11, a12, a13, a14, a15, a16, a17, a18, a19,
                           a20, a21, a22, a23, a24, a25, a26, a27, a28, a29);
```

6 Type Marshalling

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

```cpp
template <typename T>
static inline T cast_arg_uintptr(uintptr_t v) {
  using U = std::remove_reference_t<T>;
  if constexpr (std::is_pointer_v<U>) {
    return reinterpret_cast<T>(v);  // Pointers
  } else if constexpr (std::is_enum_v<U>) {
    return static_cast<T>(v);       // Enums
  } else if constexpr (std::is_integral_v<U>) {
    return static_cast<T>(v);       // Integers
  } else {
    static_assert(!sizeof(U), "Unsupported syscall arg type");
  }
}
```

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 Code Overlay and Paging System

7.1 Overview

The code overlay system enables applications of unlimited size to run within SRAM constraints. Instead of loading the entire application binary at once, the system uses a demand-loading approach where code pages are loaded into a fixed overlay window as they are needed.

7.2 Architecture

7.2.1 Overlay Window

All application code executes from a fixed overlay window at address `0x20040000`:

- **Overlay Base:** `0x20040000`
- **Overlay Size:** 256 KB
- **Page Size:** 4 KB (4096 bytes)
- **All code pages load into this fixed window**, overlaying previous pages

7.2.2 Memory Regions

The system divides SRAM into two distinct regions:

1. **Fixed Address Region** (`0x20030000+`):
   - Header section (`.app_hdr`) - must be at `0x20030000`
   - Data section (`.data`) - initialized variables at fixed addresses
   - BSS section (`.bss`) - zero-initialized data
2. **Overlay Window** (`0x20040000+`):
   - Code section (`.text`) - loaded on demand
   - Read-only data (`.rodata`) - loaded with code pages
   - Pages overlay each other - only one set active at a time

7.3 How It Works

7.3.1 Page Generation

At build time, `page_gen.py` analyzes the application binary and generates:

- Page metadata structure (`AppPageInfo`) mapping page IDs to flash offsets
- Page map array identifying which pages contain header, data, or code sections
- Section identification for proper address assignment

7.3.2 Loading Strategy

1. **Initial Load:** Header and data sections are copied immediately to fixed addresses (they're small and must be at specific locations)
2. **Critical Pages:** Code pages containing `app_setup()` and `app_loop()` are loaded before application execution begins
3. **On-Demand Loading:** When a function is called:
   - The overlay system finds which page contains the function address
   - If the page is not loaded, it copies it from flash to the overlay window
   - Previous code pages in the overlay are automatically overwritten
   - The function executes at its expected overlay address

7.3.3 Page Tracking

The code cache manager (`code_cache.cpp`) tracks:

- Which pages are currently loaded in the overlay
- Page access counts for LRU eviction (though overlay simplifies this)
- Page-to-address mapping for function pointer resolution

7.4 Benefits

- **Unlimited Application Size:** Applications can be larger than the overlay window since pages are loaded on demand
- **Simple and Stable:** Pure memory copying, no code relocation or patching required
- **Transparent:** No changes required to application code - works with any Arduino-style code
- **Efficient:** Only loads code pages that are actually executed
- **Fast Function Calls:** Code is always at the expected overlay address, enabling direct function calls

7.5 Limitations

- Only one set of code pages can be active in the overlay at a time
- Functions that call each other across page boundaries require page loading on demand
- Works best for sequential code execution patterns
- Recursive functions that span multiple pages may cause page thrashing

Overlay System Design

The overlay approach is simpler than position-independent code (PIC) or runtime relocation because:

- Code is compiled for the overlay address (`0x20040000`)
- All code pages load at the same base address
- No address patching or relocation needed
- Function pointers work correctly since addresses are stable

8 Application Binary Format

8.1 Header Structure

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

```c
typedef struct AppHeader {
  uint32_t magic;           // 'APPA' = 0x41505041
  uint32_t version;         // 0x00020000 (ABI version)
  void (*app_setup)(void);
  void (*app_loop)(void);
  SyscallGate syscall_gate; // Patched by kernel at runtime
  void* data_start;         // Start of .data section
  void* data_end;           // End of .data section
  void* bss_start;          // Start of BSS section (for zeroing)
  void* bss_end;            // End of BSS section (for zeroing)
  uint32_t* bss_marker;     // Pointer to BSS verification marker
} 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

8.2 Section Layout

```
0x20030000: +------------------+
            | .app_hdr         | Header + syscall gate (fixed)
            +------------------+
            | .data            | Initialized variables (fixed)
            +------------------+
            | .bss             | Zero-initialized data (fixed)
            +------------------+
            | __heap_start__   | Dynamic allocation
            | ...              |
            +------------------+
0x20040000: | CODE OVERLAY     | Code execution window (256 KB)
            | WINDOW           | All code pages load here
            |                  | Pages overlay each other
            +------------------+
            | ...              |
            | __heap_end__     | End of SRAM
            +------------------+
```

9 Proxy System for C++ Objects

9.1 Design Pattern

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

```cpp
struct SerialProxy {
  inline void begin(unsigned long baud) {
    Serial_begin(baud);
  }
  inline size_t print(const char* s) {
    return Serial_print_s(s);
  }
  inline void println(const char* s) {
    return Serial_println_s(s);
  }
  // ...
};

inline SerialProxy Serial;  // Global instance
```

Usage in Application:

```
Serial.begin(115200);      // Calls syscall
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

10 FreeRTOS Integration

10.1 Task Structure

The kernel uses FreeRTOS to manage application execution:

```
static void appTask(void*) {
  AppHeader* hdr = reinterpret_cast<AppHeader*>(kAppBaseAddr);
  for (;;) {
    if (hdr->app_loop != nullptr) {
      hdr->app_loop();
    }
    vTaskDelay(pdMS_TO_TICKS(1));  // Cooperative yield
  }
}
```

FreeRTOS Design Decisions

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

11 Memory Safety and Security

11.1 Current Safety Mechanisms

11.1.1 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)

11.1.2 Syscall Isolation

Syscall Isolation

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

11.1.3 Pointer Validation

The kernel validates application-provided pointers before dereferencing:

```
inline bool isValidAppPointer(const void* ptr, size_t len) {
  if (ptr == nullptr) return false;
  uintptr_t addr = reinterpret_cast<uintptr_t>(ptr);
  if (addr < kSramBaseAddr || addr >= kSramEndAddr) return false;
  if (len > 0 && addr > kSramEndAddr - len) return false;
  return true;
}
```

Pointer Validation Scope

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

11.2 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

12 Build Configuration

12.1 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`.

12.2 Linker Configuration

```
LDFLAGS:
  -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`)

13 Debugging and Diagnostics

13.1 Syscall Tracing

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

```
#define DEBUG_SYSCALLS
```

Output Format:

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

13.2 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 `build.bat`
Upload fails	Wrong board selected	Select Pico 2W in Arduino IDE

14 Performance Characteristics

14.1 Memory Usage

Memory Footprint

- **Kernel:** 50–100 KB flash (varies with Arduino core features)
- **Application Header/Data:** Typically 10–50 KB SRAM (fixed addresses)
- **Code Overlay Window:** 256 KB (for code execution, pages load on demand)
- **Available Heap:** 400 KB (remaining SRAM after header/data)
- **Application Size:** Unlimited (code pages loaded from flash as needed)

14.2 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$0.5–1.0 μs @ 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)

14.3 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)

15 Future Enhancements

15.1 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
6. **Page Preloading:** Intelligent prefetching of likely-needed pages
7. **Page Access Statistics:** Profiling tools for optimization

15.2 API Expansion

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

16 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.

17 Supported Platforms

17.1 Target Hardware

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

17.2 Development Environment

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

18 References

- Raspberry Pi Pico 2 Documentation
- Arduino RP2040 Core
- Google C++ Style Guide
- FreeRTOS Documentation

19 License

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