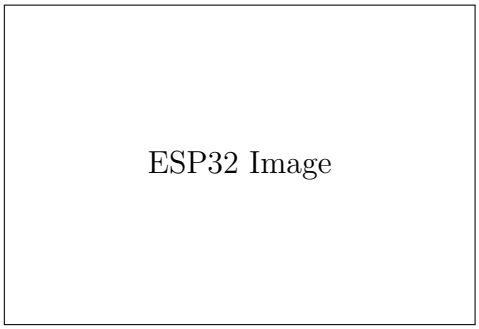
Smalltalkje

An Embedded Smalltalk for the ESP32



Based on Tim Budd's "A Little Smalltalk"

April 6, 2025

Contents

Chapter 1

Introduction

1.1 Overview

Smalltalkje is a lightweight Smalltalk implementation specifically designed for embedded systems, particularly the ESP32 microcontroller platform. It brings the power and elegance of object-oriented programming and interactive development to resource-constrained devices, making it ideal for IoT (Internet of Things) and embedded applications.

This implementation is based on Tim Budd's Little Smalltalk version 3 (Oregon State University, July 1988) but has been significantly modified and enhanced to operate effectively on embedded systems with limited memory and processing power.

1.2 Key Features

- Memory-efficient object representation
- Split memory model with objects in both RAM and Flash
- Support for various ESP32-based development boards
- Optimized bytecode interpreter
- Seamless integration with ESP32 peripherals
- WiFi connectivity
- Display support for several screen types (SSD1306 OLED, M5StickC, etc.)

1.3 Supported Hardware

Smalltalkje currently supports the following hardware platforms:

- ESP32 DevKit
- SSD1306 OLED displays
- M5StickC
- Lilygo T-Wristband
- Mac (for development)

1.4 Design Philosophy

Smalltalkje brings the power of Smalltalk to resource-constrained devices by carefully balancing the following aspects:

- **Memory Efficiency**: Using techniques like split memory model, reference counting, and optimized object representation to minimize RAM usage.
- **Execution Speed**: Optimizing the bytecode interpreter and using a method cache for improved performance.
- **Flexibility**: Maintaining the dynamic nature of Smalltalk while adapting it to embedded constraints.
- **Platform Integration**: Providing seamless access to ESP32 features like WiFi, GPIO, and displays.

The system is written in C (not C++) for the ESP-IDF framework (not Arduino) to maximize performance and minimize resource usage. This approach is essential for running Smalltalk, which traditionally requires significant memory, on resource-constrained devices.

Chapter 2

About the ESP32 and Constrained Smalltalk

2.1 The ESP32 Platform: Capabilities and Constraints

The ESP32 represents a significant advancement in microcontroller technology, offering capabilities previously unavailable in such compact and affordable devices. Designed by Espressif Systems, the ESP32 has become a cornerstone of IoT and embedded projects due to its impressive balance of features, performance, and power efficiency.

2.1.1 ESP32 Specifications

The ESP32 family of microcontrollers provides a powerful platform with the following key specifications:

- **Processor**: Dual-core Xtensa LX6 microprocessor running at up to 240 MHz
- **Memory**:
  - 520 KB to 8 MB of SRAM (depending on variant)
  - 4 MB to 16 MB of Flash memory (externally connected)
  - ROM for bootloader and core functions
- **Wireless Connectivity**:
  - WiFi (802.11 b/g/n up to 150 Mbps)

  - Bluetooth (both Classic and BLE 4.2)

- **Peripherals**:
  - Up to 34 programmable GPIO pins
  - 12-bit ADC (Analog-to-Digital Converter)
  - DAC (Digital-to-Analog Converter)
  - Multiple SPI, I2C, I2S, UART interfaces
  - PWM controllers for motor control
  - LED PWM controller
  - Hardware accelerated encryption
  - Touch sensors
- **Power Management**: Multiple power modes with current consumption ranging from 5μA in deep sleep to around 240mA during full Wi-Fi transmission

2.1.2 Constraints and Challenges

Despite its impressive specifications, the ESP32 remains a constrained device compared to desktop and server systems, presenting several challenges for running sophisticated environments like Smalltalk:

- **Memory Constraints**: While generous for a microcontroller, the ESP32's RAM is still limited for running traditional Smalltalk environments that were designed for systems with megabytes or even gigabytes of memory.
- **Execution Speed**: The ESP32's processors are significantly slower than modern desktop CPUs, requiring careful optimization of the VM for acceptable performance.
- **Flash Memory Access**: Reading from Flash is slower than RAM access, requiring thoughtful decisions about what data should reside in each memory type.
- **Power Considerations**: Many ESP32 applications run on battery power, making power consumption a crucial factor that must be addressed through efficient code and judicious use of low-power modes.

- **Limited UI Capabilities**: Most ESP32 deployments have minimal or no display capabilities, changing how a Smalltalk environment must function compared to its traditionally GUI-centric nature.
- **Real-time Requirements**: Many embedded applications have real-time constraints that traditional Smalltalk environments weren't designed to address.

2.1.3 Running Smalltalk on ESP32

Implementing a Smalltalk virtual machine on the ESP32 requires addressing several specific challenges:

- **Memory Management**: Traditional Smalltalk systems use garbage collection, which can cause unpredictable pauses. On an ESP32, a more deterministic approach is needed, leading to the choice of reference counting in Smalltalkje.
- **Object Representation**: Classical Smalltalk implementations use memory-intensive object representations. For the ESP32, a more compact representation is necessary, leading to Smalltalkje's specialized object table and split memory model.
- **Image Size**: Standard Smalltalk images can be multiple megabytes in size. For the ESP32, the image must be carefully optimized to fit within available memory while still providing useful functionality.
- **Development Environment**: Traditional Smalltalk systems include sophisticated development environments with browsers, debuggers, and inspectors. On the ESP32, a more streamlined approach is needed while still preserving Smalltalk's interactive nature.
- **Integration with Hardware**: Smalltalk must interface with ESP32's peripheral hardware, requiring well-designed primitive methods that bridge between the high-level Smalltalk environment and low-level hardware access.

Smalltalkje addresses these challenges through its innovative split memory model, reference counting garbage collection, optimized object representation, and careful integration with ESP-IDF, allowing a complete Smalltalk environment to run effectively on ESP32 devices.

2.2 Smalltalk on Constrained Devices: A Historical Perspective

While running Smalltalk on embedded devices like the ESP32 may seem ambitious, there is a rich history of implementing Smalltalk on hardware that was far more constrained than today's microcontrollers. This historical context provides valuable insights and inspiration for modern embedded Smalltalk implementations.

2.2.1 Smalltalk-80 and Early Resource Requirements

Smalltalk-80, the first widely available implementation of Smalltalk, was designed for the Xerox Alto and similar specialized hardware with the following specifications:

- 256 KB to 1 MB of RAM
- 10-20 MHz 16-bit processors
- Specialized hardware support for bit-blit operations
- Custom display hardware

These requirements were substantial for the early 1980s, leading many to view Smalltalk as requiring powerful hardware. However, several projects successfully adapted Smalltalk to run on more constrained systems.

An impressive achievement was from Softsmarts, which created a Smalltalk-80 implementation that ran on an IBM PC AT with an Intel 80286 processor running at 6 MHz with just 640 KB of RAM. This implementation demonstrated that with careful optimization, even the full Smalltalk-80 environment could run on relatively constrained hardware.

2.2.2 Smalltalk/V from Digitalk

Perhaps the most successful early constrained Smalltalk implementation was Smalltalk/V from Digitalk. This groundbreaking product ran on the original IBM PC with:

- Intel 8088 processor at 4.77 MHz
- 256 KB of RAM (though 512 KB was recommended)

- CGA graphics with 320x200 resolution and 4 colors
- MS-DOS operating system

Remarkably, these specifications are in many ways less powerful than a modern ESP32, which offers:

- Dual-core 32-bit processors at up to 240 MHz (vs. single-core 16-bit at 4.77 MHz)
- Up to 8 MB of RAM (vs. 256 KB)
- More sophisticated peripherals and connectivity options

Smalltalk/V achieved this by:

- Implementing a more compact object format
- Using a specialized memory manager
- Providing a streamlined class library
- Carefully optimizing the virtual machine for the 8088 processor

Despite these constraints, Smalltalk/V provided a complete object-oriented environment with classes, inheritance, polymorphism, and an integrated development environment with browsers and debuggers. Many developers created sophisticated applications using this constrained environment, demonstrating the inherent efficiency of well-designed Smalltalk systems.

2.2.3 Object Technology International's Embedded Smalltalk

Object Technology International (OTI, later acquired by IBM) was another pioneer in adapting Smalltalk to constrained and embedded environments. In the late 1980s and early 1990s, OTI developed specialized Smalltalk implementations for:

- Real-time embedded systems
- Industrial control applications
- Telecommunications equipment

OTI's work demonstrated that Smalltalk could be adapted for environments with:

- Hard real-time constraints
- Limited memory resources
- No graphical user interface
- Direct hardware control requirements

These embedded Smalltalk implementations influenced later technologies, including aspects of IBM's VisualAge Micro Edition and eventually Eclipse and Java technologies. The techniques developed by OTI for efficient method dispatch, compact object representation, and deterministic execution are still relevant for modern embedded Smalltalk implementations like Smalltalkje.

2.2.4 Tim Budd's Little Smalltalk

The foundation for Smalltalkje comes from Tim Budd's Little Smalltalk, a project with a remarkable history of its own. Developed by Dr. Timothy A. Budd at Oregon State University, Little Smalltalk was first created in the early 1980s as one of the earliest publicly available Smalltalk implementations.

History and Evolution

Little Smalltalk went through several major versions:

- **Version 1** (1983): The initial version, written in C, provided a simple but complete Smalltalk implementation that could run on Unix systems with modest resources.
- **Version 2** (1985-1986): Enhanced the implementation with better performance and more complete class libraries.
- **Version 3** (1987-1988): A complete rewrite that improved the object representation, enhanced the virtual machine, and expanded the class library. This is the version that serves as the foundation for Smalltalkje.
- **Version 4** (1991-1992): Added additional optimizations and features, though Smalltalkje is based on version 3 rather than this later iteration.

Technical Overview

Little Smalltalk was designed with different goals from commercial implementations like Smalltalk-80 or Smalltalk/V:

- **Educational Focus**: It was created primarily as a teaching tool to help students understand object-oriented programming concepts.
- **Simplicity**: The implementation prioritized clarity and simplicity over raw performance.
- **Portability**: Written in standard C, Little Smalltalk was designed to run on a wide variety of platforms.
- **Minimal Resource Requirements**: It was explicitly designed to run in constrained environments, making it an ideal starting point for an embedded implementation.
- **Open Source**: Perhaps most importantly for Smalltalkje, Little Smalltalk was released with source code and minimal restrictions, enabling its adaptation for new platforms like the ESP32.

Key technical aspects of Little Smalltalk version 3 include:

- A bytecode interpreter written in C
- Reference counting for memory management
- A simple but complete class hierarchy
- A parser and compiler that allowed for interactive development
- Support for basic Smalltalk concepts like blocks, inheritance, and polymorphism

From Little Smalltalk to Smalltalkje

Smalltalkje builds on the strong foundation of Little Smalltalk version 3, extending it with:

- The split memory model for efficient use of Flash and RAM
- Enhanced object representation tailored for the ESP32
- Integration with ESP32 peripherals and networking

- Additional embedded-specific classes and primitives
- Performance optimizations for the ESP32 processor

The choice of Little Smalltalk as a foundation was ideal because:

- Its clean, readable C implementation made it easy to understand and modify
- Its already minimal resource requirements provided a good starting point
- Its open source nature made adaptation legally straightforward
- Its reference counting approach to memory management was well-suited to embedded systems

The Smalltalkje project owes a tremendous debt to Tim Budd for creating and sharing Little Smalltalk. His work made it possible to bring the power and elegance of Smalltalk to a new generation of embedded devices.

Chapter 3

System Architecture

3.1 High-Level Architecture

Smalltalkje follows a modular architecture with several interacting subsystems that collectively provide a complete Smalltalk environment. The system divides into distinct layers, each with specific responsibilities.

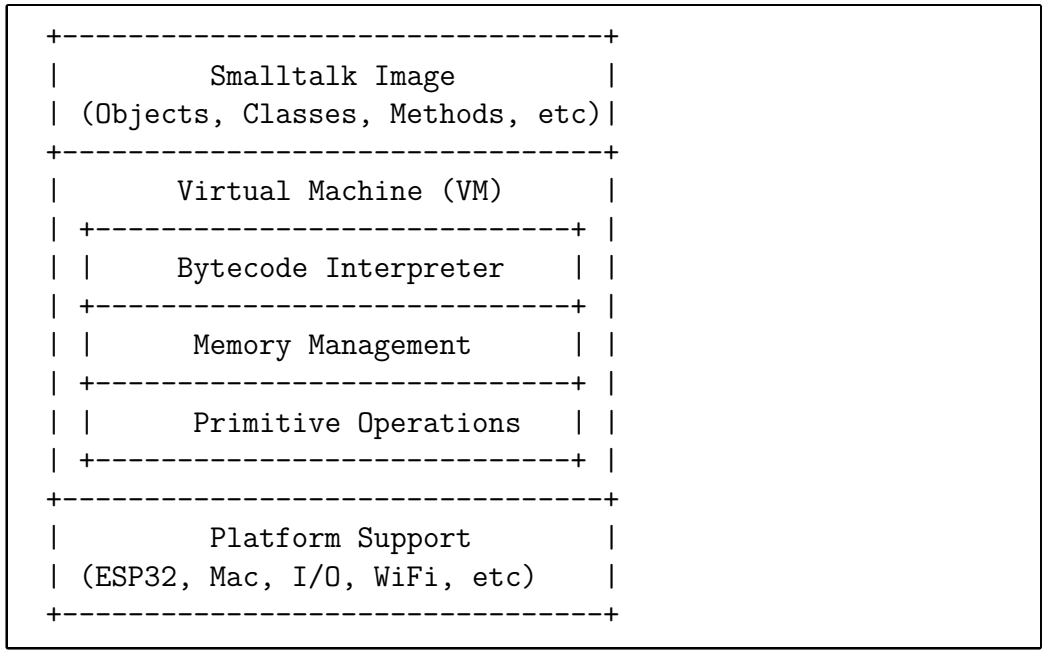


Figure 3.1: Smalltalkje System Architecture

3.2 Key Architectural Components

3.2.1 Memory Management System

The memory system in Smalltalkje is designed to operate efficiently on memory-constrained devices:

- **Object Model**: Objects are referenced via indices into an object table that holds metadata including class, size, reference count, and memory pointer
- **Reference Counting**: Simple, efficient memory reclamation using reference counting rather than more complex garbage collection
- **Split Memory Model**: Unique design that allows storing immutable objects in Flash memory while keeping mutable objects in RAM
- **Efficient Allocation**: Multi-strategy object allocation with size-specific free lists to minimize fragmentation
- **Byte Objects**: Special handling for strings, byte arrays, and other binary data through negative size values in the object table

3.2.2 Image System

The image system provides object persistence and efficient loading:

- **Smart Loading**: Can load objects selectively into RAM or leave them in Flash
- **Image Formats**: Supports both traditional (all RAM) and split (RAM/Flash) image formats
- **Object Classification**: Automatically identifies which objects can safely remain in Flash based on class (ByteArray, String, Symbol, Block)
- **Memory Optimization**: Significantly reduces RAM usage by only copying mutable objects from Flash

3.2.3 Virtual Machine Core

The VM is the heart of the system, interpreting Smalltalk bytecodes and managing execution:

- **Bytecode Interpreter**: Executes compiled Smalltalk methods with highly optimized instruction set
- **Process Management**: Time-sliced cooperative multitasking for Smalltalk processes
- **Method Cache**: Performance optimization for method lookup using a 211-entry cache
- **Context Management**: Handles method activation records and returns
- **Primitives**: Native C implementations of performance-critical operations

3.2.4 Compiler and Parser

The compilation system converts Smalltalk code to bytecodes:

- **Lexical Analysis**: Tokenization with state machine-based scanning
- **Recursive Descent Parser**: Parses method syntax and generates bytecodes
- **Method Compilation**: Optimizes common control structures and message patterns
- **Block Closure Support**: First-class functions with lexical scope

3.2.5 Platform Integration

Smalltalkje integrates closely with the underlying platform:

- **ESP32 Features**: Direct access to WiFi, HTTP, GPIO, display drivers
- **Interrupt Handling**: Bridge between ESP32 events and Smalltalk processes
- **Non-Volatile Storage**: Persistence using ESP32's NVS system
- **I/O Subsystem**: TTY, file system, and communication interfaces

Chapter 4

Virtual Machine Architecture

4.1 Object Representation

All entities in Smalltalkje are objects with a uniform representation, providing a consistent foundation for the entire system:

4.1.1 Object Table

The object table is the central registry of all objects with metadata:

- **What it does**:
  - Acts as the central registry for all objects in the system
  - Provides a level of indirection that simplifies memory management
  - Allows objects to be moved in memory without changing references
  - Maintains critical metadata for each object
  - Enables the split memory model by tracking object location
- **Components**:
  - **Index**: Object reference divided by 2, serving as the unique identifier
  - **Class reference**: Points to the class object defining the object's behavior
  - **Size**: Number of instance variables or bytes, with negative values indicating byte objects

  - **Reference count**: Tracks how many references exist to this object
  - **Memory pointer**: Points to instance variables or byte data in RAM or Flash

4.1.2 Special Object Types

Smalltalkje implements several specialized object types for different purposes:

- **Regular Objects**:
  - Store references to other objects as instance variables
  - Can mutate state by changing instance variable values
  - Reside in RAM to allow modification
  - Form the backbone of most Smalltalk programs
- **Byte Objects**:
  - Store raw bytes instead of object references
  - Represented with negative size values in the object table
  - Used for strings, byte arrays, and other binary data
  - Memory-efficient for text and binary storage
  - Access optimized for byte-level operations
- **Small Integers**:
  - Encoded directly in object references using tagged pointers
  - No object table entry needed, saving memory
  - Value stored in the upper 31 bits of the reference
  - Lowest bit set to 1 to distinguish from regular object references
  - Supports immediate arithmetic without object allocation
- **ROM Objects**:
  - Special objects stored in Flash memory
  - Immutable by design (any mutation attempt causes an error)
  - Marked with maximum reference count (0x7F) to prevent collection
  - Include strings, symbols, byte arrays, and code blocks
  - Critical for reducing RAM usage on memory-constrained devices

4.2 Bytecode Instruction Set

The VM executes compact bytecode instructions, designed specifically for efficient Smalltalk execution on embedded systems:

4.2.1 Instruction Format

- **What it does**:
  - Provides a compact encoding of operations
  - Reduces memory footprint of compiled methods
  - Enables efficient instruction decoding
  - Balances code density with execution speed
- **Components**:
  - **High nibble** (4 bits): Opcode defining the operation
  - **Low nibble** (4 bits): Operand or parameter for the operation
  - Single-byte encoding for most common operations

4.2.2 Extended Format

- **What it does**:
  - Handles operations with operands larger than 15
  - Maintains instruction set consistency
  - Provides flexibility for complex operations
- **Components**:
  - Extended opcode in the first byte
  - Additional byte(s) for larger operand values
  - Special handling by the interpreter

4.2.3 Core Instructions

The bytecode instruction set includes several categories of operations:

- **Variable Access/Store**:

  - Push/retrieve instance variables (object state)
  - Push/store temporary variables (method-local)
  - Push/store arguments (parameter passing)
  - Push literals (constants embedded in methods)
  - Direct access to self (the receiver object)
  - Special handling for super sends

- **Message Sending**:
  - Unary messages (no arguments): `object message`
  - Binary messages (one argument): `object + argument`
  - Keyword messages (multiple arguments): `object at: index put: value`
  - Special send bytecodes for common operations
  - Dynamic binding through method lookup

- **Control Flow**:
  - Conditional branches (for if/else/while)
  - Unconditional jumps (for optimization)
  - Method returns (normal, block, and non-local)
  - Block creation and activation

- **Primitive Operations**:
  - Direct calls to C-implemented operations
  - Fast paths for arithmetic, comparison, and I/O
  - Escape hatch for performance-critical code
  - Interface to platform-specific functionality

4.3 Message Passing System

Message passing is the primary mechanism for computation, implementing Smalltalk's pure object-oriented paradigm:

4.3.1 Method Lookup

- **What it does**:
  - Dynamically resolves which method to execute for a message
  - Implements Smalltalk's inheritance model
  - Provides the foundation for polymorphism
  - Handles method not found errors
- **Process**:
  - Starts search in the receiver's class
  - Traverses the inheritance chain upward
  - Returns the first matching method found
  - Invokes `doesNotUnderstand:` if no method is found
  - Results cached for performance

4.3.2 Method Cache

- **What it does**:
  - Dramatically improves performance of repeated message sends
  - Avoids costly method lookup in the class hierarchy
  - Balances memory usage with lookup speed
- **Components**:
  - 211-entry hash table (prime number size)
  - Hash based on message selector and receiver class
  - Each entry stores selector, receiver class, method class, and method
  - Invalidated when methods are modified or classes reorganized

4.3.3 Optimized Messages

- **What it does**:
  - Provides fast paths for common operations
  - Reduces overhead for frequently used messages
  - Maintains semantics while improving performance

- **Implementations**:
  - Special bytecodes for common unary messages (isNil, notNil)
  - Dedicated handling for arithmetic operations
  - Inlined implementation of simple collection access
  - Short-circuit evaluation for boolean operations

4.3.4 Primitive Messages

- **What it does**:
  - Bridges Smalltalk with native C implementation
  - Provides essential functionality impossible in pure Smalltalk
  - Optimizes performance-critical operations
  - Interfaces with hardware and platform features
- **Characteristics**:
  - Numbered primitives for basic operations
  - Class-specific primitives for specialized behavior
  - I/O and device control primitives
  - Fallback to Smalltalk code if primitive fails
  - Error reporting mechanism to Smalltalk

4.4 Memory Management

The memory manager employs several strategies for efficiency, crucial for operating in resource-constrained environments:

4.4.1 Reference Counting

- **What it does**:
  - Tracks the number of references to each object
  - Immediately reclaims memory when no references remain
  - Avoids the need for garbage collection pauses
  - Provides deterministic resource cleanup

- **Operations**:
  - Incremented when object reference is assigned
  - Decremented when reference is overwritten
  - Objects freed when count reaches zero
  - Recursively processes instance variables
  - Special handling for circularities

4.4.2 Free Lists

- **What it does**:
  - Organizes reclaimed objects by size for efficient reuse
  - Minimizes memory fragmentation
  - Reduces allocation overhead
  - Optimizes memory utilization
- **Implementation**:
  - Separate list for each object size (up to 2048 lists)
  - Free objects linked through their first instance variable
  - Quick lookup by exact size
  - Fallback strategies for size mismatches
  - Periodically compacted for efficiency

4.4.3 Block Allocation

- **What it does**:
  - Amortizes allocation overhead across multiple objects
  - Reduces memory fragmentation
  - Improves allocation speed
  - Manages memory more efficiently
- **Strategy**:
  - Allocates memory in blocks of 2048 objects
  - Distributes new objects from these blocks

  - Maintains allocation statistics
  - Reuses memory from reclaimed objects
  - Handles out-of-memory conditions gracefully

4.4.4 ROM/RAM Split

- **What it does**:
  - Dramatically reduces RAM usage on embedded devices
  - Leverages Flash memory for immutable objects
  - Preserves RAM for mutable state
  - Enables larger programs on constrained devices
- **Implementation**:
  - Immutable objects (strings, symbols, code) stored in ROM
  - Only mutable objects copied to RAM during image load
  - Object table records whether object is in ROM or RAM
  - Write-protection for ROM objects
  - Automatic classification during image creation

4.5 Context and Process Model

The VM implements a lightweight concurrency model that enables multitasking while maintaining simplicity:

4.5.1 Process Objects

- **What it does**:
  - Represents independent threads of execution
  - Enables concurrent programming model
  - Maintains execution state between time slices
  - Supports priority-based scheduling
- **Components**:
  - Priority level (determines scheduling order)

  - Link to active context (current execution point)
  - State information (running, waiting, suspended)
  - Next/previous process links (for scheduler queue)
  - Semaphore waiting links (for synchronization)

4.5.2 Context Objects

- **What it does**:
  - Stores method execution state
  - Implements the call stack
  - Provides lexical scoping for variables
  - Enables method returns and continuations
- **Components**:
  - Method reference (which method is executing)
  - Sender context (for call chain and returns)
  - Instruction pointer (current execution point)
  - Stack pointer (top of evaluation stack)
  - Arguments array (method parameters)
  - Temporary variables array (method-local state)
  - Evaluation stack (for expression calculation)

4.5.3 Stack-based Execution

- **What it does**:
  - Provides a workspace for calculating expressions
  - Passes values between operations
  - Simplifies bytecode implementation
  - Models Smalltalk expression evaluation
- **Operations**:
  - Push values onto stack (literals, variables)
  - Pop values for storage or as operation arguments

  - Duplicate or swap stack elements
  - Clear stack entries
  - Build arrays from stack elements
  - Return stack top as method result

4.5.4 Time-sliced Execution

- **What it does**:
  - Enables cooperative multitasking
  - Provides fair execution time to all processes
  - Maintains system responsiveness
  - Simulates concurrent execution
- **Implementation**:
  - Each process gets a fixed bytecode execution quota
  - Bytecode counter decremented with each instruction
  - Process yields when counter reaches zero
  - Scheduler selects next process based on priority
  - I/O operations and waits can yield immediately
  - External interrupts can preempt the current process

Chapter 5

Code Organization

5.1 Core Subsystems

5.1.1 Memory Management (memory.c)

The memory management subsystem is the foundation of Smalltalkje, responsible for object allocation, tracking, and reclamation through reference counting:

- `initMemoryManager()`: Initializes the memory system by setting up the object table, clearing free list pointers, zeroing reference counts, and building initial free lists. This must be called before any other memory operations can be performed.
- `allocObject()`: Implements a sophisticated multi-strategy allocation algorithm for efficient object creation:
  1. First attempts to find an exact-sized object in the free list (fastest path)
  2. Tries to repurpose a size-0 object by expanding it to the needed size
  3. Finds a larger object and uses it (potentially wasting some space)
  4. Locates a smaller object, frees its memory, and resizes it

  This approach maximizes memory reuse while minimizing fragmentation.
- `allocByte()`: Creates specialized objects for storing raw bytes (like strings and byte arrays) rather than object references. These have negative size fields to indicate their special nature.

- `sysDecr()`: The core of the reference counting system that reclaims objects when their reference count reaches zero. It:
  1. Validates the reference count isn't negative (error detection)
  2. Decrements the reference count of the object's class
  3. Adds the object to the appropriate free list
  4. Recursively decrements the reference count of all instance variables
  5. Clears all instance variables to prevent dangling references
- `visit()`: Rebuilds reference counts during image loading by implementing a depth-first traversal of the object graph. This ensures only reachable objects are retained in memory.
- `mBlockAlloc()`: Manages memory allocation in large blocks (2048 objects at a time) rather than individual malloc calls. This amortizes allocation overhead across many objects and reduces memory fragmentation.

5.1.2 Interpreter (interp.c)

The interpreter is the heart of the Smalltalk VM, executing bytecodes and implementing the dynamic message dispatch system:

- `execute()`: The main bytecode execution loop, which operates as a large state machine processing one bytecode at a time. It:
  1. Extracts execution state from the process object (stack, context, etc.)
  2. Runs a loop decoding and executing bytecodes until the time slice ends
  3. Handles message sends, primitive calls, returns, and control flow
  4. Implements time-sliced cooperative multitasking via bytecode counting
  5. Saves execution state back to the process when yielding
- `findMethod()`: Implements Smalltalk's inheritance-based method lookup by searching for a matching method starting from a class and proceeding up the inheritance hierarchy. This is the core of dynamic dispatch.

- `flushCache()`: Invalidates method cache entries when methods have been recompiled or modified, ensuring the cached version is no longer used.
- Method cache optimization: The VM uses a 211-entry cache to significantly improve performance by avoiding repeated method lookups. Each cache entry stores:
  - The message selector being sent
  - The class of the receiver
  - The class where the method was found
  - The actual method object

5.1.3 Image Management (image.c)

The image management subsystem handles object persistence and the unique split memory model of Smalltalkje:

- `imageRead()`: Loads a complete traditional Smalltalk image where all objects are placed in RAM. It:
  1. Reads the symbols table reference (root object)
  2. Processes each object's metadata (index, class, size)
  3. Allocates memory for and loads each object's data
  4. Restores reference counts and rebuilds free lists
- `readTableWithObjects()`: Implements the memory-optimized split approach where:
  1. All object table entries are loaded into RAM
  2. Immutable objects (ByteArray, String, Symbol, Block) point directly to Flash memory
  3. Mutable objects are copied into RAM

  This critical optimization significantly reduces RAM usage on ESP32 devices by keeping large portions of the image in Flash.
- `writeObjectTable()`: Saves only the metadata for objects (index, class, size, flags) to a file. It identifies and marks ROM-eligible objects (immutable types) via flags, enabling the split memory optimization when the image is later loaded.

- `writeObjectData()`: Writes the actual content of all objects to a separate file. In the split memory approach, this data can be embedded in Flash memory and accessed directly for immutable objects.

5.1.4 Parser (parser.c)

The parser translates Smalltalk source code into bytecodes, implementing a complete compiler for method definitions:

- `parse()`: The main entry point that coordinates the entire parsing process from method selector to method body. It populates a Method object with:
  - Message selector (method name)
  - Bytecodes (executable instructions)
  - Literals (constants used in the method)
  - Stack and temporary variable size information
  - Optionally the source text for debugging
- Recursive descent parsing: The parser implements Smalltalk grammar through a set of mutually recursive functions, each handling specific language constructs:
  - `messagePattern()`: Parses method selectors (unary, binary, keyword)
  - `temporaries()`: Handles temporary variable declarations
  - `body()`: Processes the sequence of statements in a method
  - `statement()`: Handles individual statements including returns
  - `expression()`: Parses complex expressions including assignments
  - `term()`: Processes basic expression elements (variables, literals, blocks)
  - `block()`: Handles block closure syntax and semantics
- Control flow optimizations: The parser intelligently transforms common control structures into efficient bytecode sequences:
  - `ifTrue:`/`ifFalse:` become conditional branch instructions
  - `whileTrue:` becomes an optimized loop structure
  - `and:`/`or:` implement short-circuit evaluation

  - These optimizations avoid the overhead of message sends for these common patterns

- Variable scoping: The parser correctly resolves variable references across different scopes:
  - `self`/`super` (receiver references)
  - Temporary variables (method-local)
  - Method arguments
  - Instance variables (object state)
  - Global variables (looked up at runtime)
- Bytecode generation: The parser produces compact bytecode instructions through:
  - `genInstruction()`: Creates instructions with high nibble (opcode) and low nibble (operand)
  - `genLiteral()`: Manages the literal table for constants used in methods
  - `genCode()`: Appends raw bytecodes to the instruction stream
- Block closure handling: The parser implements Smalltalk's powerful block closures (anonymous functions with lexical scope) by:
  1. Creating a Block object with metadata about arguments
  2. Generating code to invoke the block creation primitive at runtime
  3. Compiling the block body inline but skipping it during normal execution
  4. Handling special block return semantics

5.1.5 Lexical Analysis (lex.c)

The lexical analyzer tokenizes Smalltalk source code, converting raw text into a stream of tokens for the parser to process:

- State machine architecture: The lexer implements a finite state machine with different states for:
  - Regular token recognition (identifiers, keywords, operators)
  - String literal processing (with escape sequences)

  - Numeric literal parsing (integers and floating point)
  - Comment handling (single and multi-line)
  - Special character processing (brackets, parentheses, etc.)

- Token types: The lexer identifies and categorizes different elements of Smalltalk syntax:
  - `nameconst`: Identifiers like variable names
  - `namecolon`: Keywords in method selectors (ending with colon)
  - `binary`: Binary operators and other special characters
  - `intconst`: Integer literals
  - `floatconst`: Floating-point literals
  - `charconst`: Character literals
  - `symconst`: Symbol literals (starting with #)
  - `strconst`: String literals (in single quotes)
  - `arraybegin`: Start of literal arrays (#( ))
  - Miscellaneous tokens for punctuation and syntax elements

- Advanced features:
  - Look-ahead capability for multi-character operators and tokens
  - Proper handling of nested comments
  - Support for various numeric formats (decimal, hex, scientific notation)
  - Proper string escape sequence processing
  - Error detection for malformed tokens

- The lexer maintains state between calls to provide a continuous token stream to the parser, with functions like:
  - `nextToken()`: Advances to and returns the next token
  - `lexinit()`: Initializes the lexer with source text
  - `peek()`: Examines the next character without consuming it
  - Error reporting with precise location information

5.1.6 ESP32 Integration

The ESP32 integration layer bridges the Smalltalk environment with the powerful capabilities of the ESP32 microcontroller:

- **esp32wifi.c**: Implements complete WiFi networking functionality for Smalltalk:
  - Network scanning, connection, and management
  - IP address and network status handling
  - Event-driven connectivity with callbacks to Smalltalk methods
  - Support for both station and access point modes
  - Integration with FreeRTOS task management and event handling
- **esp32http.c**: Provides HTTP client capabilities that allow Smalltalk programs to:
  - Make GET, POST, PUT, and DELETE requests to web servers
  - Handle headers, request bodies, and response parsing
  - Process chunked transfer encoding and redirects
  - Implement REST API clients and web service integrations
  - Support both blocking and non-blocking request patterns
- **esp32nvs.c**: Implements a non-volatile storage interface that enables:
  - Persistent storage of Smalltalk objects between reboots
  - Key-value style storage in the ESP32's NVS flash partition
  - Different namespaces for organizing related data
  - Storage of strings, integers, and binary data from Smalltalk
  - Recovery of application state after power loss
- **esp32io.c**: Provides general-purpose I/O control, exposing capabilities like:
  - GPIO pin manipulation (digital read/write, PWM output)
  - Analog-to-digital conversion for sensor readings
  - Hardware timers and interrupts with Smalltalk callbacks
  - Low-level peripheral control (I2C, SPI, UART configuration)
  - Power management and deep sleep functionality

5.1.7 Display Support

Smalltalkje includes sophisticated display support for several screen types, enabling graphical user interfaces:

- SSD1306 OLED driver integration:
  - Full control of monochrome 128x64 and 128x32 OLED displays
  - Pixel-level drawing operations and text rendering
  - Buffer-based graphics with efficient partial updates
  - I2C communication with configurable address and pins
  - Low-level display commands for advanced control
- M5StickC display support:
  - Complete integration with M5StickC's color LCD
  - Hardware-accelerated drawing via ESP32's SPI capabilities
  - Font rendering with multiple built-in typefaces
  - Sprite and bitmap handling for fluid animations
  - Touch input integration for interactive applications
  - Power-efficient display management
- Graphics primitives for drawing:
  - Lines, rectangles, circles, and polygons
  - Bitmap rendering and scaling
  - Text with multiple font options
  - Compositing operations (XOR, OR, AND modes)
  - Screen buffering for flicker-free updates
  - Coordinate transformations and clipping

Chapter 6

Smalltalk Class Hierarchy

Smalltalkje implements a comprehensive class hierarchy that provides all the core functionality of Smalltalk. The class hierarchy is based on Tim Budd's Little Smalltalk, but has been extended and optimized for embedded systems.

6.1 Core Classes

- **Object**: The root class of the entire class hierarchy. Every class inherits from Object, which provides basic functionality common to all objects:
  - Identity operations (==, =, hash)
  - Type checking (isKindOf:, isMemberOf:)
  - Copying (copy, deepCopy, shallowCopy)
  - Printing and display (printString, print, display)
  - Basic collection protocols (do:, collect:, select:)
- **Class**: Represents the behavior and structure of objects in the system:
  - Instance creation (new, new:)
  - Method management (addMethod:, removeMethod:)
  - Class hierarchy navigation (superClass, subClasses)
  - Introspection capabilities (methods, respondsTo:)
- **Boolean** and its subclasses **True** and **False**:
  - Conditional testing (ifTrue:, ifFalse:, ifTrue:ifFalse:)

  - Logical operations (and:, or:, not, xor:)

- **UndefinedObject**: The class of nil, representing the absence of a value:
  - Special handling for nil-checking (isNil, ifNil:, ifNotNil:)
- **Block**: First-class functions with lexical scope:
  - Evaluation with arguments (value, value:, value:value:, etc.)
  - Control structures (whileTrue:, whileFalse:)

6.2 Collection Classes

A rich hierarchy of collection classes provides flexible data structures:

Collection Abstract base class for all collections

IndexedCollection Collections with numeric indices

Array Fixed-size ordered collection
ByteArray Array of bytes
String Array of characters (subclass of ByteArray)
Dictionary Key-value mapping

List Growable linked list

Set Collection with no duplicates (subclass of List)

Interval Sequence of values with step

6.3 Other Important Classes

- **Context**: Represents method execution contexts, forming the call stack:
  - Method arguments and temporary variables storage
  - Call stack linkage (sender context)
  - Block return handling
- **Integer**: Fixed-precision integer values:
  - Arithmetic operations (+, -, *, /, etc.)

  - Comparison operations (<, >, <=, >=, etc.)
  - Bit manipulation (bitAnd:, bitOr:, bitShift:)
- **Method**: Represents compiled methods:
  - Bytecode storage
  - Literals table
  - Method execution
  - Source text storage (optional)
- **Symbol**: Unique string-like objects used for method selectors and keys:
  - Identity-based equality (==) for fast comparisons
  - Used in the method dictionary
  - Special handling in the image system
- **Link**: Building block for linked data structures:
  - Key-value storage
  - Next link reference
  - Used to implement List and Dictionary classes

6.4 ESP32-Specific Classes

Smalltalkje includes several ESP32-specific classes for interacting with the hardware:

- **WiFi**: Interface to ESP32 WiFi capabilities
- **GPIO**: Digital and analog I/O control
- **HTTP**: Web client functionality
- **NVS**: Non-volatile storage access
- **Display**: Interface to connected displays
- **Timer**: Hardware timer access

6.5 Class Implementation

Classes in Smalltalkje are defined in plain Smalltalk syntax in .st files. For example, this excerpt from basic.st shows the definition of core classes:

```
Class Object
Class Block Object context argCount argLoc bytePointer
Class Boolean Object
Class    True Boolean
Class    False Boolean
Class Class Object name instanceSize methods superClass
   variables
Class Context Object linkLocation method arguments
   temporaries
Class Integer Object
```

Method definitions follow, organized by classes and protocol categories:

```
Methods Boolean 'all'
  ifTrue: trueBlock
    ^ self ifTrue: trueBlock ifFalse: []
|
  ifFalse: falseBlock
    ^ self ifTrue: [] ifFalse: falseBlock
|
  ifFalse: falseBlock ifTrue: trueBlock
    ^ self ifTrue: trueBlock
      ifFalse: falseBlock
|
  and: aBlock
    ^ self ifTrue: aBlock ifFalse: [ false ]
|
  or: aBlock
    ^ self ifTrue: [ true ] ifFalse: aBlock
```

Chapter 7

Smalltalk Syntax

Smalltalk uses a simple, elegant syntax that enables powerful object-oriented programming. The language is designed to be readable and expressive, with a focus on message passing as the primary computation mechanism.

7.1 Basic Syntax Elements

- **Comments**: Enclosed between quote marks "..."

```
"This is a comment in Smalltalk"
```

- **Variables**: Start with a lowercase letter, can contain alphanumeric characters

```
counter
myVariable
x1
```

- **Class Names**: Start with an uppercase letter

```
Object
Collection
Array
```

- **Literals**:
  - **Numbers**: Integer and floating point

```
42
3.14159
```

  - **Strings**: Enclosed in single quotes

```
'Hello, world!'
'Embedded Smalltalk'
```

  - **Characters**: Preceded by a dollar sign

```
$a
$5
$\n
```

  - **Symbols**: Preceded by a hash sign

```
#mySymbol
#at:put:
```

  - **Arrays**: Enclosed in hash parentheses

```
#(1 2 3 4)
#('a' 'b' 'c')
```

- **Statement Terminator**: Period (.) separates statements

```
x := 1. y := 2. z := x + y.
```

- **Assignment**: Uses := operator

```
counter := 0
result := 42
```

- **Return**: Uses caret symbol (^) operator

```
^ self size * 2
^ 'Result: ', value printString
```

7.2 Message Sending

In Smalltalk, all computation happens through message sending. Messages come in three forms:

- **Unary Messages**: No arguments

```
object size
collection isEmpty
number negated
```

- **Binary Messages**: One argument, operator-like syntax

```
3 + 4
string , ' world'
a < b
```

- **Keyword Messages**: One or more arguments, named parameters

```
collection at: 1
dictionary at: #key put: value
array copyFrom: 1 to: 10
```

Message precedence (from highest to lowest):

1. Unary messages
2. Binary messages
3. Keyword messages

Parentheses can be used to override precedence:

```
(3 + 4) squared
(array at: index) printString
```

7.3 Control Structures

Smalltalk implements control structures as messages sent to objects, primarily blocks (closures):

- **Conditionals**:

```
(x > 0) ifTrue: [ 'positive' ] ifFalse: [ 'non-
   positive' ]
object isNil ifTrue: [ object := self defaultValue ]
```

- **Loops**:

```
[condition] whileTrue: [ actions ]
[counter < 10] whileTrue: [ counter := counter + 1 ]
```

- **Iteration**:

```
collection do: [:each | each doSomething]
1 to: 10 do: [:i | sum := sum + i]
```

7.4 Block Closures

Blocks (delimited by square brackets) are anonymous functions with lexical scope:

```
[:x | x * x]
[self doSomething]
[:a :b | a + b]
```

Blocks can be assigned to variables and passed as arguments:

```
square := [:x | x * x].
result := square value: 5.   (* 25 *)

adder := [:a :b | a + b].
sum := adder value: 3 value: 4.   (* 7 *)
```

7.5 Class and Method Definition

Classes are defined using a special syntax:

```
Class NewClass SuperClass instVar1 instVar2 instVar3
```

Methods are organized in categories and defined as follows:

```
Methods ClassName 'category'
  methodName
    "Method comment"
    | temp1 temp2 |
    statements.
    ^ returnValue
|
  anotherMethod: arg1 with: arg2
    "Method comment"
    statements.
    ^ returnValue
```

7.6 Common Idioms

- **Collection Manipulation**:

```
collection select: [:each | each isEven]
collection collect: [:each | each asString]
```

```
collection inject: 0 into: [:sum :each | sum + each]
```

- **Error Handling**:

```
self error: 'Invalid input'
[riskOperation] on: Error do: [:error | handleError]
```

- **Resource Management**:

```
file := File open: 'data.txt'.
[file processContents] ensure: [file close]
```

Chapter 8

Memory Optimization Details

Smalltalkje employs several innovative techniques to minimize RAM usage on embedded devices, which is critical for running on resource-constrained ESP32 systems.

8.1 Split Memory Model

The most significant memory optimization is the split memory approach:

1. **Object Table**: Always loaded into RAM for fast access to metadata
2. **Object Data**: Selectively distributed between RAM and Flash:
    - Mutable objects (regular objects) copied to RAM
    - Immutable objects (strings, symbols, bytearrays, blocks) kept in Flash

8.2 ROM-Eligible Objects

Specific classes are identified as immutable and can safely remain in Flash:

- ByteArray (class 18)
- String (class 34)
- Symbol (class 8)
- Block (class 182)

These objects have their reference count set to the maximum (0x7F) to prevent garbage collection, and their memory pointers point directly to Flash memory regions.

8.3 Memory Allocation Strategies

Object allocation follows a multi-strategy approach for efficiency:

1. **Free list exact match**: Reuse object of exactly the right size
2. **Zero-size expansion**: Repurpose a size-0 object with new memory
3. **Larger object reuse**: Repurpose larger object (potentially wasting some space)
4. **Smaller object reuse**: Reallocate memory for smaller object

8.4 Block Allocation

Rather than individual malloc calls for each object, Smalltalkje allocates memory in blocks of 2048 object slots, significantly reducing allocation overhead and fragmentation.

8.5 Image Loading Optimization

The image loading process is optimized to:

- Read object table and object data separately
- Identify ROM-eligible objects during image saving
- Point ROM objects directly to Flash memory
- Only copy mutable objects to RAM
- Rebuild reference counts only for live objects

This approach enables much larger Smalltalk programs to run on the ESP32 than would be possible with a traditional all-RAM image.

Chapter 9

Building and Using Smalltalkje

9.1 Prerequisites

To build and use Smalltalkje, you'll need:

- ESP-IDF (Espressif IoT Development Framework)
- ESP32 toolchain
- ESP32 development board (DevKit, M5StickC, etc.)
- USB cable for connecting to the ESP32
- C compiler (GCC)

For development on Mac:

- Xcode or command-line tools
- CMake

9.2 Building for ESP32

1. Clone the repository:

```
git clone https://github.com/aknabi/smalltalkje.git
cd smalltalkje
```

2. Configure the build for your ESP32 board in sdkconfig

```
make menuconfig
```

3. Build the project:

```
make
```

4. Flash the build to your ESP32:

```
make flash
```

5. Monitor serial output:

```
make monitor
```

9.3 Development Workflow

When developing with Smalltalkje on the ESP32:

1. Write Smalltalk code (.st files) for your application
2. Build and load these into the Smalltalk image
3. Compile the image into a form optimized for ESP32
4. Flash the system with the new image
5. Test and debug using the interactive Smalltalk environment

9.4 Interactive Development

Smalltalkje provides a REPL (Read-Eval-Print Loop) for interactive development:

```
st> 3 + 4
7
st> Collection allSubclasses
(Array List Set Dictionary IndexedCollection ByteArray
   String)
st> 1 to: 10 do: [:i | i print]
12345678910
```

9.5 Building Custom Applications

To build a custom application:

1. Create your application classes in .st files
2. Add them to the image building process (in smalltalkImage/Makefile)
3. Define a startup class or method for your application
4. Build the image and flash to the ESP32

Example of a simple application class:

```
Class WeatherStation Object temperature humidity
   lastUpdate

Methods WeatherStation 'initialization'
  initialize
    temperature := 0.
    humidity := 0.
    lastUpdate := nil.
    ^ self
|
  readSensors
    (* Read values from connected sensors *)
    temperature := DHT11 readTemperature.
    humidity := DHT11 readHumidity.
    lastUpdate := DateTime now.
    ^ self
|
  displayReadings
    Display clear.
    Display at: 0 at: 0 print: 'Temp: ', temperature
   printString, 'C'.
    Display at: 0 at: 10 print: 'Humidity: ', humidity
   printString, '%'.
    Display at: 0 at: 20 print: lastUpdate printString.
    ^ self
```

Chapter 10

Conclusion

Smalltalkje brings the power and elegance of Smalltalk to embedded systems, particularly the ESP32 platform. By carefully optimizing memory usage, implementing a split memory model, and providing seamless integration with the ESP32's capabilities, Smalltalkje enables sophisticated object-oriented applications on resource-constrained devices.

The system provides a complete Smalltalk environment with:

- Full object-oriented programming model
- Rich class library
- Interactive development environment
- Efficient bytecode interpreter
- Memory-optimized object representation
- Deep integration with ESP32 capabilities

Smalltalkje demonstrates that even complex, dynamic languages can be adapted to work effectively on embedded systems, opening up new possibilities for IoT and embedded application development. By combining the flexibility of Smalltalk with the power of the ESP32, developers can create sophisticated applications with less code and faster development cycles than traditional embedded programming approaches.