

DAP SDK Test Framework - Complete Guide

Async Testing, Mocking, and Test Automation

Cellframe Development Team

October 28, 2025

Contents

1	Document Information	3
1.1	Revision History	3
1.2	Copyright	3
1.3	License	3
2	Part I: Introduction	4
2.1	1. Overview	4
2.1.1	1.1 What is DAP SDK Test Framework?	4
2.1.2	1.2 Why Use This Framework?	4
2.1.3	1.3 Key Features at a Glance	4
2.1.4	1.4 Quick Comparison	4
2.1.5	1.5 Target Audience	5
2.1.6	1.6 Prerequisites	5
2.2	2. Quick Start	6
2.2.1	2.1 Your First Test (5 minutes)	6
2.2.2	2.2 Adding Async Timeout (2 minutes)	6
2.2.3	2.3 Adding Mocks (5 minutes)	7
2.3	3. API Reference	9
2.3.1	3.1 Async Testing API	9
2.3.2	3.2 Mock Framework API	10
2.3.3	3.3 Custom Linker Wrapper API	13
2.3.4	3.4 CMake Integration	14
2.3.5	3.5 Async Mock Execution	16
2.4	4. Complete Examples	19
2.4.1	4.1 State Machine Test (Real Project Example)	19
2.4.2	4.2 Mock with Callback	20
2.4.3	4.3 Mock with Execution Delays	20
2.4.4	4.4 Custom Linker Wrapper (Advanced)	21
2.4.5	4.5 Dynamic Mock Behavior	22
2.4.6	4.6 Mocking Functions in Static Libraries	23
2.4.7	4.7 Asynchronous Mock Execution	25
2.5	5. Glossary	27
2.6	6. Troubleshooting	28
2.6.1	Issue: Test Hangs Indefinitely	28
2.6.2	Issue: High CPU	28
2.6.3	Issue: Mock Not Called (Real Function Executes)	28
2.6.4	Issue: Wrong Return Value	28

2.6.5 Issue: Flaky Tests (Intermittent Failures)	28
2.6.6 Issue: Compilation Error “undefined reference to __wrap”	29
2.6.7 Issue: Mock Callback Not Executing	29
2.6.8 Issue: Mock Not Working for Functions in Static Library	29
2.6.9 Issue: Linker Error “multiple definition”	30
2.6.10 Issue: Delay Not Working	30

1 Document Information

Version: 1.0.1

Date: October 28, 2025

Status: Production Ready

Language: English

1.1 Revision History

Version	Date	Changes	Author
1.0.1	2025-10-28	Updated examples, improved API reference, added troubleshooting	Cellframe Team
1.0.0	2025-10-27	Initial comprehensive guide	Cellframe Team

1.2 Copyright

Copyright © 2025 Demlabs. All rights reserved.

This document describes the DAP SDK Test Framework, part of the Cellframe Network project.

1.3 License

See project LICENSE file for terms and conditions.

2 Part I: Introduction

2.1 1. Overview

The DAP SDK Test Framework is a production-ready testing infrastructure designed for the Cellframe blockchain ecosystem. It provides comprehensive tools for testing asynchronous operations, mocking external dependencies, and ensuring reliable test execution across platforms.

2.1.1 1.1 What is DAP SDK Test Framework?

A complete testing solution that includes:

- **Async Testing Framework** - Tools for testing asynchronous operations with timeouts
- **Mock Framework** - Function mocking without code modification
- **Async Mock Execution** - Asynchronous mock callbacks with thread pool
- **Auto-Wrapper System** - Automatic linker configuration
- **Self-Tests** - 21 test functions validating framework reliability

2.1.2 1.2 Why Use This Framework?

Problem: Testing asynchronous code is hard - Operations complete at unpredictable times - Network delays vary - Tests can hang indefinitely - External dependencies complicate testing

Solution: This framework provides - [x] Timeout protection (global + per-operation) - [x] Efficient waiting (polling + condition variables) - [x] Dependency isolation (mocking) - [x] Realistic simulation (delays, failures) - [x] Thread-safe operations - [x] Cross-platform support

2.1.3 1.3 Key Features at a Glance

Feature	Description	Benefit
Global Timeout	alarm + siglongjmp	Prevents CI/CD hangs
Condition Polling	Configurable intervals	Efficient async waiting
pthread Helpers	Condition variable wrappers	Thread-safe coordination
Mock Framework	Linker-based (- -wrap)	Zero technical debt
Async Mocks	Thread pool execution	Real async behavior simulation
Delays	Fixed, Range, Variance	Realistic timing simulation
Callbacks	Inline + Runtime	Dynamic mock behavior
Auto-Wrapper	Bash/PowerShell scripts	Automatic setup
Self-Tests	21 test functions	Validated reliability

2.1.4 1.4 Quick Comparison

Traditional Approach:

```
// [!] Busy waiting, no timeout, CPU waste
while (!done) {
    usleep(10000); // 10ms sleep
}
```

With DAP Test Framework:

```
// [+] Efficient, timeout-protected, automatic logging
DAP_TEST_WAIT_UNTIL(done == true, 5000, "Should complete");
```

2.1.5 1.5 Target Audience

- DAP SDK developers
- Cellframe SDK contributors
- VPN Client developers
- Anyone testing async C code in Cellframe ecosystem

2.1.6 1.6 Prerequisites

Required Knowledge: - C programming - Basic understanding of async operations
- CMake basics - pthread concepts (for advanced features)

Required Software: - GCC 7+ or Clang 10+ (or MinGW on Windows) - CMake 3.10+
- pthread library - Linux, macOS, or Windows (partial support)

2.2 2. Quick Start

2.2.1 2.1 Your First Test (5 minutes)

Step 1: Create test file

```
// my_test.c
#include "dap_test.h"
#include "dap_common.h"

#define LOG_TAG "my_test"

int main() {
    dap_common_init("my_test", NULL);

    // Test code
    int result = 2 + 2;
    dap_assert_PIF(result == 4, "Math should work");

    log_it(L_INFO, "[+] Test passed!");

    dap_common_deinit();
    return 0;
}
```

Step 2: Create CMakeLists.txt

```
add_executable(my_test my_test.c)
target_link_libraries(my_test dap_core)
add_test(NAME my_test COMMAND my_test)
```

Step 3: Build and run

```
cd build
cmake ..
make my_test
./my_test
```

2.2.2 2.2 Adding Async Timeout (2 minutes)

```
#include "dap_test.h"
#include "dap_test_async.h"
#include "dap_common.h"

#define LOG_TAG "my_test"
#define TIMEOUT_SEC 30

int main() {
    dap_common_init("my_test", NULL);

    // Add global timeout
    dap_test_global_timeout_t timeout;
    if (dap_test_set_global_timeout(&timeout, TIMEOUT_SEC, "My Test")) {
        return 1; // Timeout triggered
    }
}
```

```

    // Your tests here

    dap_test_cancel_global_timeout();
    dap_common_deinit();
    return 0;
}

```

Update CMakeLists.txt:

```

# Link test framework library (includes dap_test, dap_mock, etc.)
target_link_libraries(my_test dap_test dap_core pthread)

```

2.2.3 2.3 Adding Mocks (5 minutes)

```

#include "dap_test.h"
#include "dap_mock.h"
#include "dap_common.h"
#include <assert.h>

#define LOG_TAG "my_test"

// Declare mock
DAP MOCK_DECLARE(external_api_call);

int main() {
    dap_common_init("my_test", NULL);
    // Note: dap_mock_init() not needed - auto-initialized!

    // Configure mock to return 42
    DAP MOCK_SET_RETURN(external_api_call, (void*)42);

    // Run code that calls external_api_call
    int result = my_code_under_test();

    // Verify mock was called once and returned correct value
    assert(DAP MOCK_GET_CALL_COUNT(external_api_call) == 1);
    assert(result == 42);

    log_it(L_INFO, "[+] Test passed!");

    // Optional cleanup (if you need to reset mocks)
    // dap_mock_deinit();
    dap_common_deinit();
    return 0;
}

```

Update CMakeLists.txt:

```

include(${CMAKE_CURRENT_SOURCE_DIR}/../test-framework/mocks/DAPMockAutoWrap.cmak

# Link test framework library (includes dap_test, dap_mock, etc.)
target_link_libraries(my_test dap_test dap_core pthread)

```

```
# Auto-generate --wrap linker flags  
dap_mock_utowrap(my_test)  
  
# If you need to mock functions in static libraries:  
# dap_mock_utowrap_with_static(my_test dap_static_lib)
```


2.3 3. API Reference

2.3.1 3.1 Async Testing API

2.3.1.1 Global Timeout

```
int dap_test_set_global_timeout(  
    dap_test_global_timeout_t *a_timeout,  
    uint32_t a_timeout_sec,  
    const char *a_test_name  
);  
// Returns: 0 on setup, 1 if timeout triggered  
  
void dap_test_cancel_global_timeout(void);
```

2.3.1.2 Condition Polling

```
bool dap_test_wait_condition(  
    dap_test_condition_cb_t a_condition,  
    void *a_user_data,  
    const dap_test_async_config_t *a_config  
);  
// Returns: true if condition met, false on timeout  
//  
// Callback signature:  
// typedef bool (*dap_test_condition_cb_t)(void *a_user_data);  
//  
// Config structure:  
// typedef struct {  
//     uint32_t timeout_ms;           // Max wait time (ms)  
//     uint32_t poll_interval_ms;    // Polling interval (ms)  
//     bool fail_on_timeout;         // abort() on timeout?  
//     const char *operation_name;   // For logging  
// } dap_test_async_config_t;  
//  
// Default config: DAP_TEST_ASYNC_CONFIG_DEFAULT  
// - timeout_ms: 5000 (5 seconds)  
// - poll_interval_ms: 100 (100 ms)  
// - fail_on_timeout: true  
// - operation_name: "async operation"
```

2.3.1.3 pthread Helpers

```
void dap_test_cond_wait_init(dap_test_cond_wait_ctx_t *a_ctx);  
bool dap_test_cond_wait(dap_test_cond_wait_ctx_t *a_ctx, uint32_t a_timeout_ms);  
void dap_test_cond_signal(dap_test_cond_wait_ctx_t *a_ctx);  
void dap_test_cond_wait_deinit(dap_test_cond_wait_ctx_t *a_ctx);
```

2.3.1.4 Time Utilities

```
uint64_t dap_test_get_time_ms(void); // Monotonic time in ms  
void dap_test_sleep_ms(uint32_t a_delay_ms); // Cross-platform sleep
```

2.3.1.5 Macros

```
DAP_TEST_WAIT_UNTIL(condition, timeout_ms, msg)
// Quick inline condition waiting
```

2.3.2 3.2 Mock Framework API

Header: dap_mock.h

2.3.2.1 Framework Initialization

```
int dap_mock_init(void);
// Optional: Reinitialize mock framework (auto-initialized via constructor)
// Returns: 0 on success
// Note: Framework auto-initializes before main(), manual call not required
// Cross-platform: Uses __attribute__((constructor)) on GCC/Clang/MinGW,
//                  static C++ object on MSVC

void dap_mock_deinit(void);
// Cleanup mock framework (call in teardown if needed)
// Note: Also auto-deinitializes async system if enabled
// Auto-cleanup: Uses __attribute__((destructor)) on GCC/Clang,
//               atexit() on MSVC for automatic cleanup after main()
```

2.3.2.2 Mock Declaration Macros Simple Declaration (auto-enabled, return 0):

```
DAP MOCK_DECLARE(function_name);
```

With Configuration Structure:

```
DAP MOCK_DECLARE(function_name, {
    .enabled = true,
    .return_value.l = 0xDEADBEEF,
    .delay = {
        .type = DAP MOCK_DELAY_FIXED,
        .fixed_us = 1000
    }
});
```

With Inline Callback:

```
DAP MOCK_DECLARE(function_name, {.return_value.i = 0}, {
    // Callback body - custom logic for each call
    if (a_arg_count >= 1) {
        int arg = (int)(intptr_t)a_args[0];
        return (void*)(intptr_t)(arg * 2); // Double the input
    }
    return (void*)0;
});
```

For Custom Wrapper (no auto-wrapper generation):

```
DAP MOCK_DECLARE_CUSTOM(function_name, {
    .delay = {
```

```

        .type = DAP MOCK_DELAY_VARIANCE,
        .variance = {.center_us = 100000, .variance_us = 50000}
    }
});

```

2.3.2.3 Configuration Structures `dap_mock_config_t`:

```

typedef struct dap_mock_config {
    bool enabled; // Enable/disable mock
    dap_mock_return_value_t return_value; // Return value
    dap_mock_delay_t delay; // Execution delay
    bool async; // Execute callback asynchronously (default)
    bool call_original_before; // Call original function BEFORE mock log
    bool call_original_after; // Call original function AFTER mock log
} dap_mock_config_t;

// Default: enabled=true, return=0, no delay, sync, no original call
#define DAP MOCK_CONFIG_DEFAULT { \
    .enabled = true, \
    .return_value = {0}, \
    .delay = {.type = DAP MOCK_DELAY_NONE}, \
    .async = false, \
    .call_original_before = false, \
    .call_original_after = false \
}

// Passthrough config: track calls but always call original before mock (for int)
#define DAP MOCK_CONFIG_PASSTHROUGH { \
    .enabled = true, \
    .return_value = {0}, \
    .delay = {.type = DAP MOCK_DELAY_NONE}, \
    .async = false, \
    .call_original_before = true, \
    .call_original_after = false \
}

```

`dap_mock_return_value_t`:

```

typedef union dap_mock_return_value {
    int i; // For int, bool, small types
    long l; // For pointers (cast with intptr_t)
    uint64_t u64; // For uint64_t, size_t (64-bit)
    void *ptr; // For void*, generic pointers
    char *str; // For char*, strings
} dap_mock_return_value_t;

```

`dap_mock_delay_t`:

```

typedef enum {
    DAP MOCK_DELAY_NONE, // No delay
    DAP MOCK_DELAY_FIXED, // Fixed delay
    DAP MOCK_DELAY_RANGE, // Random in [min, max]
    DAP MOCK_DELAY_VARIANCE // Center ± variance
} dap_mock_delay_type_t;

```

```

typedef struct dap_mock_delay {
    dap_mock_delay_type_t type;
    union {
        uint64_t fixed_us;
        struct { uint64_t min_us; uint64_t max_us; } range;
        struct { uint64_t center_us; uint64_t variance_us; } variance;
    };
} dap_mock_delay_t;

```

2.3.2.4 Control Macros

```

DAP MOCK_ENABLE(func_name)
// Enable mock (intercept calls)
// Example: DAP MOCK_ENABLE(dap_stream_write);

DAP MOCK_DISABLE(func_name)
// Disable mock (call real function)
// Example: DAP MOCK_DISABLE(dap_stream_write);

DAP MOCK_RESET(func_name)
// Reset call history and statistics
// Example: DAP MOCK_RESET(dap_stream_write);

DAP MOCK_SET_RETURN(func_name, value)
// Set return value (cast with (void*) or (void*)(intptr_t))
// Example: DAP MOCK_SET_RETURN(dap_stream_write, (void*)(intptr_t)42);

DAP MOCK_GET_CALL_COUNT(func_name)
// Get number of times mock was called (returns int)
// Example: int count = DAP MOCK_GET_CALL_COUNT(dap_stream_write);

DAP MOCK_WAS_CALLED(func_name)
// Returns true if called at least once (returns bool)
// Example: assert(DAP MOCK_WAS_CALLED(dap_stream_write));

DAP MOCK_GET_ARG(func_name, call_idx, arg_idx)
// Get specific argument from a specific call
// call_idx: 0-based index of call (0 = first call)
// arg_idx: 0-based index of argument (0 = first argument)
// Returns: void* (cast to appropriate type)
// Example: void *buffer = DAP MOCK_GET_ARG(dap_stream_write, 0, 1);
//          size_t size = (size_t)DAP MOCK_GET_ARG(dap_stream_write, 0, 2);

```

2.3.2.5 Delay Configuration Macros

```

DAP MOCK_SET_DELAY_FIXED(func_name, microseconds)
DAP MOCK_SET_DELAY_MS(func_name, milliseconds)
// Set fixed delay

DAP MOCK_SET_DELAY_RANGE(func_name, min_us, max_us)
DAP MOCK_SET_DELAY_RANGE_MS(func_name, min_ms, max_ms)

```

```
// Set random delay in range

DAP MOCK_SET_DELAY_VARIANCE(func_name, center_us, variance_us)
DAP MOCK_SET_DELAY_VARIANCE_MS(func_name, center_ms, variance_ms)
// Set delay with variance (e.g., 100ms ± 20ms)

DAP MOCK_CLEAR_DELAY(func_name)
// Remove delay
```

2.3.2.6 Callback Configuration

```
DAP MOCK_SET_CALLBACK(func_name, callback_func, user_data)
// Set custom callback function

DAP MOCK_CLEAR_CALLBACK(func_name)
// Remove callback (use return_value instead)

// Callback signature:
typedef void* (*dap_mock_callback_t)(
    void **a_args,
    int a_arg_count,
    void *a_user_data
);
```

2.3.3 3.3 Custom Linker Wrapper API

Header: dap_mock_linker_wrapper.h

2.3.3.1 DAP MOCK WRAPPER_CUSTOM Macro Creates custom linker wrapper with PARAM syntax:

```
DAP MOCK_WRAPPER_CUSTOM(return_type, function_name,
    PARAM(type1, name1),
    PARAM(type2, name2),
    ...
) {
    // Custom wrapper implementation
}
```

Features: - Automatically generates function signature - Automatically creates void* argument array with proper casts - Automatically checks if mock is enabled - Automatically executes configured delay - Automatically records call - Calls real function if mock disabled

Example:

```
DAP MOCK_WRAPPER_CUSTOM(int, my_function,
    PARAM(const char*, path),
    PARAM(int, flags),
    PARAM(mode_t, mode)
) {
    // Your custom logic here
    if (strcmp(path, "/dev/null") == 0) {
```

```

        return -1; // Simulate error
    }
    return 0; // Success
}

```

PARAM Macro: - Format: PARAM(type, name) - Extracts type and name automatically - Handles casting to void* correctly - Uses uintptr_t for safe casting of pointers and integer types

2.3.3.2 Simpler Wrapper Macros For common return types:

```

DAP MOCK WRAPPER INT(func_name, (params), (args))
DAP MOCK WRAPPER PTR(func_name, (params), (args))
DAP MOCK WRAPPER VOID_FUNC(func_name, (params), (args))
DAP MOCK WRAPPER BOOL(func_name, (params), (args))
DAP MOCK WRAPPER SIZE_T(func_name, (params), (args))

```

2.3.4 3.4 CMake Integration

CMake Module: mocks/DAPMockAutoWrap.cmake

```
include(${CMAKE_SOURCE_DIR}/dap-sdk/test-framework/mocks/DAPMockAutoWrap.cmake)
```

```
# Automatically scan sources and generate --wrap flags
```

```
dap_mock_autowrap(target_name)
```

```
# Alternative: specify source files explicitly
```

```
dap_mock_autowrap(TARGET target_name SOURCE file1.c file2.c)
```

How it works: 1. Scans source files for DAP MOCK DECLARE patterns 2. Extracts function names 3. Adds -Wl, --wrap=function_name to linker flags 4. Works with GCC, Clang, MinGW

2.3.4.1 Mocking Functions in Static Libraries Problem: When linking static libraries (lib*.a), functions may be excluded from the final executable if they are not directly used. This causes --wrap flags to not work for functions inside static libraries.

Solution: Use the dap_mock_autowrap_with_static() function to wrap static libraries with --whole-archive flags, which forces the linker to include all symbols from the static library.

Usage Example:

```
include(${CMAKE_SOURCE_DIR}/dap-sdk/test-framework/mocks/DAPMockAutoWrap.cmake)
```

```
add_executable(test_http_client
    test_http_client.c
    test_http_client_mocks.c
)
```

```
# Normal linking
```

```
target_link_libraries(test_http_client
    dap_test # Test framework
```

```

    dap_core          # Core library
    dap_http_server   # Static library to mock
    pthread
)

# Auto-generate --wrap flags from test sources
dap_mock_autowrap(test_http_client)

# Important: wrap static library with --whole-archive AFTER dap_mock_autowrap!
# This forces linker to include all symbols from dap_http_server,
# including those only used internally
dap_mock_autowrap_with_static(test_http_client dap_http_server)

```

What `dap_mock_autowrap_with_static` does: 1. Rebuilds the link libraries list 2. Wraps specified static libraries with flags: - `-Wl,--whole-archive` (before library) - `<library_name>` (the library itself) - `-Wl,--no-whole-archive` (after library) 3. Adds `-Wl,--allow-multiple-definition` to handle duplicate symbols

Important Notes:

1. Order of calls matters:

```

# Correct:
dap_mock_autowrap(test_target)           # First auto-generation
dap_mock_autowrap_with_static(test_target lib) # Then --whole-archive

# Incorrect:
dap_mock_autowrap_with_static(test_target lib) # This overwrites previous
dap_mock_autowrap(test_target)

```

2. Multiple libraries:

```

# Can wrap multiple static libraries at once
dap_mock_autowrap_with_static(test_target
    dap_http_server
    dap_stream
    dap_crypto
)

```

3. Limitations:

- Works only with GCC, Clang, and MinGW
- May increase executable size
- Do not use for shared libraries (`.so`, `.dll`)

Complete Configuration Example:

```

include(${CMAKE_SOURCE_DIR}/dap-sdk/test-framework/mocks/DAPMockAutoWrap.cmake)

add_executable(test_stream_mocks
    test_stream_mocks.c
    test_stream_mocks_wrappers.c
)

target_link_libraries(test_stream_mocks
    dap_test

```

```

dap_stream      # Static library
dap_net         # Static library
dap_core
pthread
)

target_include_directories(test_streammocks PRIVATE
    ${CMAKE_SOURCE_DIR}/dap-sdk/test-framework
    ${CMAKE_SOURCE_DIR}/dap-sdk/core/include
)

# Auto-generate --wrap flags
dap_mock_autowrap(test_streammocks)

# Wrap static libraries for mocking internal functions
dap_mock_autowrap_with_static(test_streammocks
    dap_stream
    dap_net
)

```

Verifying Configuration:

```

# Check linker flags
cd build
make VERBOSE=1 | grep -E "--wrap|--whole-archive"

# Should see:
# -Wl,--wrap=dap_stream_write
# -Wl,--wrap=dap_net_tun_create
# -Wl,--whole-archive ... dap_stream ... -Wl,--no-whole-archive
# -Wl,--whole-archive ... dap_net ... -Wl,--no-whole-archive

```

2.3.5 3.5 Async Mock Execution

Header: dap_mock_async.h

Provides lightweight asynchronous execution for mock callbacks without requiring full dap_events infrastructure. Perfect for unit tests that need to simulate async behavior in isolation.

2.3.5.1 Initialization

```

// Initialize async system with worker threads
int dap_mock_async_init(uint32_t a_worker_count);
// a_worker_count: 0 = auto, typically 1-2 for unit tests
// Returns: 0 on success

// Deinitialize (waits for all pending tasks)
void dap_mock_async_deinit(void);

// Check if initialized
bool dap_mock_async_is_initialized(void);

```


2.3.5.2 Task Scheduling

```
// Schedule async callback execution
dap_mock_async_task_t* dap_mock_async_schedule(
    dap_mock_async_callback_t a_callback,
    void *a_arg,
    uint32_t a_delay_ms // 0 = immediate
);

// Cancel pending task
bool dap_mock_async_cancel(dap_mock_async_task_t *a_task);
```

2.3.5.3 Waiting for Completion

```
// Wait for specific task
bool dap_mock_async_wait_task(
    dap_mock_async_task_t *a_task,
    int a_timeout_ms // -1 = infinite, 0 = no wait
);

// Wait for all pending tasks
bool dap_mock_async_wait_all(int a_timeout_ms);
// Returns: true if all completed, false on timeout
```

2.3.5.4 Async Mock Configuration To enable async execution for a mock, set `.async = true` in config:

```
// Async mock with delay
DAP MOCK_DECLARE_CUSTOM(dap_client_http_request, {
    .enabled = true,
    .async = true, // Execute callback asynchronously
    .delay = {
        .type = DAP MOCK_DELAY_FIXED,
        .fixed_us = 50000 // 50ms
    }
});

// Mock wrapper (executes asynchronously if dap_mock_async_init() was called)
DAP MOCK_WRAPPER_CUSTOM(void, dap_client_http_request,
    PARAM(const char*, a_url),
    PARAM(callback_t, a_callback),
    PARAM(void*, a_arg)
) {
    // This code runs in worker thread after delay
    a_callback("response data", 200, a_arg);
}
```

2.3.5.5 Utility Functions

```
// Get pending task count
size_t dap_mock_async_get_pending_count(void);
```

```

// Get completed task count
size_t dap_mock_async_get_completed_count(void);

// Execute all pending tasks immediately (fast-forward time)
void dap_mock_async_flush(void);

// Reset statistics
void dap_mock_async_reset_stats(void);

// Set default delay for async mocks
void dap_mock_async_set_default_delay(uint32_t a_delay_ms);

```

2.3.5.6 Usage Pattern

```

void test_async_http(void) {
    // Note: No manual init needed! Async system auto-initializes with mock fram

    volatile bool done = false;

    // Call function with async mock (configured with .async = true)
    dap_client_http_request("http://test.com", callback, &done);

    // Wait for async completion
    DAP_TEST_WAIT_UNTIL(done, 5000, "HTTP request");

    // Or wait for all async mocks
    bool completed = dap_mock_async_wait_all(5000);
    assert(completed && done);

    // Cleanup (optional, handled by dap_mock_deinit())
    // dap_mock_deinit(); // Auto-cleans async system too
}

```

Note: Async system is automatically initialized when mock framework starts (via constructor). Manual `dap_mock_async_init()` only needed if you want to customize worker count.

2.4 4. Complete Examples

2.4.1 4.1 State Machine Test (Real Project Example)

Example from cellframe-srv-vpn-client/tests/unit/test_vpn_state_handlers.c:

```
#include "dap_test.h"
#include "dap_mock.h"
#include "vpn_state_machine.h"
#include "vpn_state_handlers_internal.h"

#define LOG_TAG "test_vpn_state_handlers"

// Declare mocks with simple configuration
DAP MOCK_DECLARE(dap_net_tun_deinit);
DAP MOCK_DECLARE(dap_chain_node_client_close_mt);
DAP MOCK_DECLARE(vpn_wallet_close);

// Mock with return value configuration
DAP MOCK_DECLARE(dap_chain_node_client_connect_mt, {
    .return_value.l = 0xDEADBEEF
});

static vpn_sm_t *s_test_sm = NULL;

static void setup_test(void) {
    // Note: dap_mock_init() auto-called, not needed here
    s_test_sm = vpn_sm_init();
    assert(s_test_sm != NULL);
}

static void teardown_test(void) {
    if (s_test_sm) {
        vpn_sm_deinit(s_test_sm);
        s_test_sm = NULL;
    }
    // Optional: dap_mock_deinit() to reset mocks between tests
}

void test_state_disconnected_cleanup(void) {
    log_it(L_INFO, "TEST: state_disconnected_entry() cleanup");

    setup_test();

    // Setup state with resources
    s_test_sm->tun_handle = (void*)0x12345678;
    s_test_sm->wallet = (void*)0xABCDEF00;
    s_test_sm->node_client = (void*)0x22222222;

    // Enable mocks
    DAP MOCK_ENABLE(dap_net_tun_deinit);
    DAP MOCK_ENABLE(vpn_wallet_close);
    DAP MOCK_ENABLE(dap_chain_node_client_close_mt);
```

```

// Call state handler
state_disconnected_entry(s_test_sm);

// Verify cleanup was performed
assert(DAP MOCK_GET_CALL_COUNT(dap_net_tun_deinit) == 1);
assert(DAP MOCK_GET_CALL_COUNT(vpn_wallet_close) == 1);
assert(DAP MOCK_GET_CALL_COUNT(dap_chain_node_client_close_mt) == 1);

teardown_test();
log_it(L_INFO, "[+] PASS");
}

int main() {
    dap_common_init("test_vpn_state_handlers", NULL);

    test_state_disconnected_cleanup();

    log_it(L_INFO, "All tests PASSED [OK]");
    dap_common_deinit();
    return 0;
}

```

2.4.2 4.2 Mock with Callback

```

#include "dap_mock.h"

DAP MOCK_DECLARE(dap_hash_fast, {.return_value.i = 0}, {
    if (a_arg_count >= 2) {
        uint8_t *data = (uint8_t*)a_args[0];
        size_t size = (size_t)a_args[1];
        uint32_t hash = 0;
        for (size_t i = 0; i < size; i++) {
            hash += data[i];
        }
        return (void*)(intptr_t)hash;
    }
    return (void*)0;
});

void test_hash() {
    uint8_t data[] = {1, 2, 3};
    uint32_t hash = dap_hash_fast(data, 3);
    assert(hash == 6); // Callback sums bytes
}

```

2.4.3 4.3 Mock with Execution Delays

Example from `dap-sdk/net/client/test/test_http_client_mocks.h`:

```

#include "dap_mock.h"

```

```

// Mock with variance delay: simulates realistic network jitter
// 100ms ± 50ms = range of 50-150ms
#define HTTP_CLIENT MOCK_CONFIG_WITH_DELAY ((dap_mock_config_t){ \
    .enabled = true, \
    .delay = { \
        .type = DAP_MOCK_DELAY_VARIANCE, \
        .variance = { \
            .center_us = 100000, /* 100ms center */ \
            .variance_us = 50000 /* ±50ms variance */ \
        } \
    } \
})

// Declare mock with simulated network latency
DAP_MOCK_DECLARE_CUSTOM(dap_client_http_request_full,
                        HTTP_CLIENT_MOCK_CONFIG_WITH_DELAY);

// Mock without delay for cleanup operations (instant execution)
DAP_MOCK_DECLARE_CUSTOM(dap_client_http_close_unsafe, {
    .enabled = true,
    .delay = {.type = DAP_MOCK_DELAY_NONE}
});

```

2.4.4 4.4 Custom Linker Wrapper (Advanced)

Example from test_http_client_mocks.c using DAP_MOCK_WRAPPER_CUSTOM:

```

#include "dap_mock.h"
#include "dap_mock_linker_wrapper.h"
#include "dap_client_http.h"

// Declare mock (registers with framework)
DAP_MOCK_DECLARE_CUSTOM(dap_client_http_request_async,
                        HTTP_CLIENT_MOCK_CONFIG_WITH_DELAY);

// Custom wrapper implementation with full control
// DAP_MOCK_WRAPPER_CUSTOM generates:
// - __wrap_dap_client_http_request_async function signature
// - void* args array for mock framework
// - Automatic delay execution
// - Call recording
DAP_MOCK_WRAPPER_CUSTOM(void, dap_client_http_request_async,
    PARAM(dap_worker_t*, a_worker),
    PARAM(const char*, a_uplink_addr),
    PARAM(uint16_t, a_uplink_port),
    PARAM(const char*, a_method),
    PARAM(const char*, a_path),
    PARAM(dap_client_http_callback_full_t, a_response_callback),
    PARAM(dap_client_http_callback_error_t, a_error_callback),
    PARAM(void*, a_callbacks_arg)
) {
    // Custom mock logic - simulate async HTTP behavior
}

```

```

// This directly invokes callbacks based on mock configuration

if (g_mock_http_response.should_fail && a_error_callback) {
    // Simulate error response
    a_error_callback(g_mock_http_response.error_code, a_callbacks_arg);
} else if (a_response_callback) {
    // Simulate successful response with configured data
    a_response_callback(
        g_mock_http_response.body,
        g_mock_http_response.body_size,
        g_mock_http_response.headers,
        a_callbacks_arg,
        g_mock_http_response.status_code
    );
}
// Note: Configured delay is executed automatically before this code
}

```

CMakeLists.txt:

```

# Include auto-wrap helper
include(${CMAKE_SOURCE_DIR}/dap-sdk/test-framework/mocks/DAPMockAutoWrap.cmake)

add_executable(test_http_client
    test_http_client_mocks.c
    test_http_client_mocks.h
    test_main.c
)

target_link_libraries(test_http_client
    dap_test      # Test framework with mocks
    dap_core      # DAP core library
    pthread       # Threading support
)

# Auto-generate --wrap linker flags by scanning all sources
dap_mock_autowrap(test_http_client)

```

2.4.5 4.5 Dynamic Mock Behavior

```

// Mock that changes behavior based on call count
// Simulates flaky network: fails first 2 times, then succeeds
DAP MOCK_DECLARE(flaky_network_send, {.return_value.i = 0}, {
    int call_count = DAP MOCK_GET_CALL_COUNT(flaky_network_send);

    // Fail first 2 calls (simulate network issues)
    if (call_count < 2) {
        log_it(L_DEBUG, "Simulating network failure (attempt %d)", call_count +
            return (void*)(intptr_t)-1; // Error code
    }

    // Succeed on 3rd and subsequent calls

```

```

    log_it(L_DEBUG, "Network call succeeded");
    return (void*)(intptr_t)0; // Success code
});

void test_retry_logic() {
    // Test function that retries on failure
    int result = send_with_retry(data, 3); // Max 3 retries

    // Should succeed on 3rd attempt
    assert(result == 0);
    assert(DAP MOCK_GET_CALL_COUNT(flaky_network_send) == 3);

    log_it(L_INFO, "[+] Retry logic works correctly");
}

```

2.4.6 4.6 Mocking Functions in Static Libraries

Example test that mocks functions inside static library dap_stream:

CMakeLists.txt:

```

include(${CMAKE_SOURCE_DIR}/dap-sdk/test-framework/mocks/DAPMockAutoWrap.cmake)

add_executable(test_stream_mocks
    test_stream_mocks.c
    test_stream_mocks_wrappers.c
)

target_link_libraries(test_stream_mocks
    dap_test
    dap_stream      # Static library - need to mock functions inside
    dap_net
    dap_core
    pthread
)

target_include_directories(test_stream_mocks PRIVATE
    ${CMAKE_SOURCE_DIR}/dap-sdk/test-framework
    ${CMAKE_SOURCE_DIR}/dap-sdk/core/include
)

# Step 1: Auto-generate --wrap flags from test sources
dap_mock_autowrap(test_stream_mocks)

# Step 2: Wrap static library with --whole-archive
# This forces linker to include all symbols from dap_stream,
# including internal functions that need to be mocked
dap_mock_autowrap_with_static(test_stream_mocks dap_stream)

test_stream_mocks.c:
#include "dap_test.h"
#include "dap_mock.h"

```

```

#include "dap_stream.h"
#include "dap_common.h"
#include <assert.h>

#define LOG_TAG "test_stream_mocks"

// Mock function used inside dap_stream
DAP MOCK_DECLARE(dap_net_tun_write, {
    .return_value.i = 0, // Success
    .delay = {
        .type = DAP MOCK_DELAY_FIXED,
        .fixed_us = 10000 // 10ms delay
    }
});

// Wrap function for mocking
DAP MOCK_WRAPPER_CUSTOM(int, dap_net_tun_write,
    PARAM(int, a_fd),
    PARAM(const void*, a_buf),
    PARAM(size_t, a_len)
) {
    // Mock logic - simulate successful write
    log_it(L_DEBUG, "Mock: dap_net_tun_write called (fd=%d, len=%zu)", a_fd, a_len);
    return 0;
}

void test_stream_write_with_mock(void) {
    log_it(L_INFO, "TEST: Stream write with mocked tun_write");

    // Create stream (dap_stream uses dap_net_tun_write internally)
    dap_stream_t *stream = dap_stream_create(...);
    assert(stream != NULL);

    // Perform write - should use mocked dap_net_tun_write
    int result = dap_stream_write(stream, "test data", 9);

    // Verify mock was called
    assert(result == 0);
    assert(DAP MOCK_GET_CALL_COUNT(dap_net_tun_write) > 0);

    dap_stream_delete(stream);
    log_it(L_INFO, "[+] Test passed");
}

int main() {
    dap_common_init("test_stream_mocks", NULL);

    test_stream_write_with_mock();

    dap_common_deinit();
    return 0;
}

```



```
}
```

Key Points: 1. `dap_mock_utowrap()` must be called **before** `dap_mock_utowrap_with_static`
2. Specify all static libraries where functions need to be mocked 3. `--whole-`
archive may increase executable size 4. Works only with GCC, Clang, and
MinGW

2.4.7 4.7 Asynchronous Mock Execution

Example demonstrating async mock callbacks with thread pool:

```
#include "dap_mock.h"  
#include "dap_test_async.h"
```

```
// Async mock for HTTP request with 50ms delay  
DAP MOCK_DECLARE_CUSTOM(dap_client_http_request, {  
    .enabled = true,  
    .async = true, // Execute in worker thread  
    .delay = {  
        .type = DAP MOCK_DELAY_FIXED,  
        .fixed_us = 50000 // 50ms realistic network latency  
    }  
});
```

```
// Mock wrapper - executes asynchronously  
DAP MOCK_WRAPPER_CUSTOM(int, dap_client_http_request,  
    PARAM(const char*, a_url),  
    PARAM(http_callback_t, a_callback),  
    PARAM(void*, a_arg)  
) {  
    // This code runs in worker thread after 50ms delay  
    const char *response = "{\"status\":\"ok\",\"data\":\"test\"}";  
    a_callback(response, 200, a_arg);  
    return 0;  
}
```

```
static volatile bool s_callback_executed = false;  
static volatile int s_http_status = 0;
```

```
static void http_response_callback(const char *body, int status, void *arg) {  
    s_http_status = status;  
    s_callback_executed = true;  
    log_it(L_INFO, "HTTP response received: status=%d", status);  
}
```

```
void test_async_http_request(void) {  
    log_it(L_INFO, "TEST: Async HTTP request");
```

```
    s_callback_executed = false;  
    s_http_status = 0;
```

```

// Call HTTP request - mock will execute asynchronously
int result = dap_client_http_request(
    "http://test.com/api",
    http_response_callback,
    NULL
);

assert(result == 0);
log_it(L_DEBUG, "HTTP request initiated, waiting for callback...");

// Wait for async mock to complete (up to 5 seconds)
DAP_TEST_WAIT_UNTIL(s_callback_executed, 5000, "HTTP callback");

// Verify
assert(s_callback_executed);
assert(s_http_status == 200);

// Alternative: wait for all async mocks
bool all_completed = dap_mock_async_wait_all(5000);
assert(all_completed);

log_it(L_INFO, "[+] Async mock test passed");
}

// Fast-forward example: test without real delays
void test_async_with_flush(void) {
    s_callback_executed = false;

    // Schedule async task with long delay
    dap_client_http_request("http://test.com", http_response_callback, NULL);

    // Instead of waiting 50ms, execute immediately
    dap_mock_async_flush(); // Fast-forward time

    // Callback already executed
    assert(s_callback_executed);

    log_it(L_INFO, "[+] Fast-forward test passed");
}

```

Benefits of Async Mocks: - Realistic simulation of network/IO latency - No need for full `dap_events` infrastructure in unit tests - Thread-safe execution - Deterministic testing with `flush()` - Statistics tracking with `get_pending_count()` / `get_completed_count()`

2.5 5. Glossary

Async Mock - Mock callback executing in separate thread with configurable delay

Async Operation - Operation completing at unpredictable future time

Auto-Wrapper - System auto-generating linker - -wrap flags from source

Callback - Function pointer executed on event

Condition Polling - Repeatedly checking condition until met or timeout

Condition Variable - pthread primitive for thread synchronization

Constructor Attribute - GCC attribute running function before main()

Designated Initializers - C99 struct init: {.field = value}

Global Timeout - Time limit for entire test suite via SIGALRM

Linker Wrapping - -wrap=func redirects calls to __wrap_func

Mock - Fake function implementation for testing

Monotonic Clock - Time source unaffected by system time changes

Poll Interval - Time between condition checks

pthread - POSIX threads library

Return Value Union - Tagged union for type-safe mock returns

Self-Test - Test validating the testing framework itself

siglongjmp/sigsetjmp - Signal-safe non-local jump

Thread Pool - Set of worker threads for async task execution

Thread-Safe - Works correctly with concurrent access

Timeout - Maximum wait time before giving up

Union - C type holding different types in same memory

2.6 6. Troubleshooting

2.6.1 Issue: Test Hangs Indefinitely

Symptom: Test runs forever without completing

Cause: Async operation never signals completion

Solution: Add global timeout protection

```
dap_test_global_timeout_t timeout;
if (dap_test_set_global_timeout(&timeout, 30, "Tests")) {
    log_it(L_ERROR, "Test timeout!");
}
```

Prevention: Always use DAP_TEST_WAIT_UNTIL with reasonable timeout

2.6.2 Issue: High CPU

Symptom: 100% CPU during test

Solution: Increase poll interval or use pthread helpers

```
cfg.poll_interval_ms = 500; // Less frequent polling
```

2.6.3 Issue: Mock Not Called (Real Function Executes)

Symptom: Real function executes instead of mock

Cause: Missing linker -wrap flag

Solution: Verify CMake configuration and linker flags

```
# Check if linker flags are present
make VERBOSE=1 | grep -- "--wrap"
```

```
# Should see: -Wl,--wrap=function_name
```

Fix: Ensure `dap_mock_autowrap(target)` is called after `add_executable()`

2.6.4 Issue: Wrong Return Value

Symptom: Mock returns unexpected value

Solution: Use correct union field

```
.return_value.i = 42 // int
.return_value.l = 0xDEAD // pointer
.return_value.ptr = ptr // void*
```

2.6.5 Issue: Flaky Tests (Intermittent Failures)

Symptom: Sometimes pass, sometimes fail

Cause: Race conditions, insufficient timeouts, or timing assumptions

Solution: Increase timeouts and add tolerance for timing-sensitive checks

```
// For network operations - use generous timeout
cfg.timeout_ms = 60000; // 60 sec for network operations
```

```
// For timing checks - use tolerance range
uint64_t elapsed = measure_time();
```

```
assert(elapsed >= 90 && elapsed <= 150); // ±50ms tolerance

// Use variance delay for realistic simulation
DAP MOCK_SET_DELAY_VARIANCE(func, 100000, 50000); // 100ms ± 50ms
```

2.6.6 Issue: Compilation Error “undefined reference to __wrap”

Symptom: Linker error about __wrap_function_name

Solution: Ensure dap_mock_utowrap() is called in CMakeLists.txt

```
include(${CMAKE_SOURCE_DIR}/dap-sdk/test-framework/mocks/DAPMockAutoWrap.cmake)
dap_mock_utowrap(my_test)
```

2.6.7 Issue: Mock Callback Not Executing

Symptom: Mock returns configured value, but callback logic doesn’t run

Cause: Callback not registered or mock disabled

Solution: Verify callback is set and mock is enabled

```
// Declare with inline callback (preferred)
DAP MOCK_DECLARE(func_name, {.enabled = true}, {
    // Your callback logic here
    return (void*)42;
});
```

```
// Or set callback at runtime
DAP MOCK_SET_CALLBACK(func_name, my_callback, user_data);
```

```
// Ensure mock is enabled
DAP MOCK_ENABLE(func_name);
```

Note: Callback return value overrides .return_value configuration

2.6.8 Issue: Mock Not Working for Functions in Static Library

Symptom: Functions from static library (lib*.a) are not mocked, real function executes

Cause: Linker excludes unused symbols from static libraries, so --wrap is not applied

Solution: Use dap_mock_utowrap_with_static() to wrap static library with --whole-archive flags

```
# After normal linking and dap_mock_utowrap()
dap_mock_utowrap(test_target)
```

```
# Wrap static library with --whole-archive
dap_mock_utowrap_with_static(test_target dap_http_server)
```

Verify:

```
make VERBOSE=1 | grep -E "--whole-archive|dap_http_server"
# Should see: -Wl,--whole-archive ... dap_http_server ... -Wl,--no-whole-archive
```

Important: Order matters! First dap_mock_utowrap(), then dap_mock_utowrap_with_static()

2.6.9 Issue: Linker Error “multiple definition”

Symptom: Error multiple definition of 'function_name' when using --whole-archive

Cause: Some symbols are defined in multiple libraries

Solution: dap_mock_autowrap_with_static() automatically adds --allow-multiple-definition, but if issue persists:

Explicitly add flag

target_link_options(test_target PRIVATE "-Wl,--allow-multiple-definition")

Alternative: Use --whole-archive only for specific libraries that require mocking

2.6.10 Issue: Delay Not Working

Symptom: Mock executes instantly despite delay config

Solution: Verify delay is set after mock declaration

DAP MOCK_DECLARE(func_name);

DAP MOCK_SET_DELAY_MS(func_name, 100); *// Set after declare*