

Robot Framework Boardfarm Integration - Development Plan

Project: robotframework-boardfarm

Goal: Integrate Boardfarm testbed with Robot Framework as test execution and reporting engine

Reference: pytest-boardfarm (existing pytest integration)

Created: January 17, 2025

Executive Summary

This plan outlines the development of `robotframework-boardfarm`, a library that integrates the Boardfarm testbed framework with Robot Framework. The integration follows the same architectural patterns as `pytest-boardfarm`, adapting them to Robot Framework's listener-based extension model and keyword-driven testing paradigm.

Analysis Summary

Existing pytest-boardfarm Integration Pattern

The pytest-boardfarm integration provides:

1. **Plugin Registration** via pytest entry points (`pytest11`)
2. **Command Line Arguments** forwarded to Boardfarm's pluggy hooks
3. **Fixtures** for device access (`device_manager` , `boardfarm_config` , `bf_context` , `bf_logger`)
4. **Lifecycle Management** via pytest hooks:
 - `pytest_sessionstart` → Device reservation and config parsing
 - `pytest_runtestloop` → Device deployment and teardown
 - `pytest_runtest_setup` → Environment requirement validation
5. **HTML Report Integration** via pytest-html hooks
6. **Markers** for environment requirements (`@pytest.mark.env_req`)

Boardfarm Core Architecture

Boardfarm uses:

- **Pluggy** for hook-based plugin system
- **DeviceManager** for device lifecycle management
- **BoardfarmConfig** for merged inventory/environment configuration
- **Async deployment** via `asyncio.run()` for environment setup

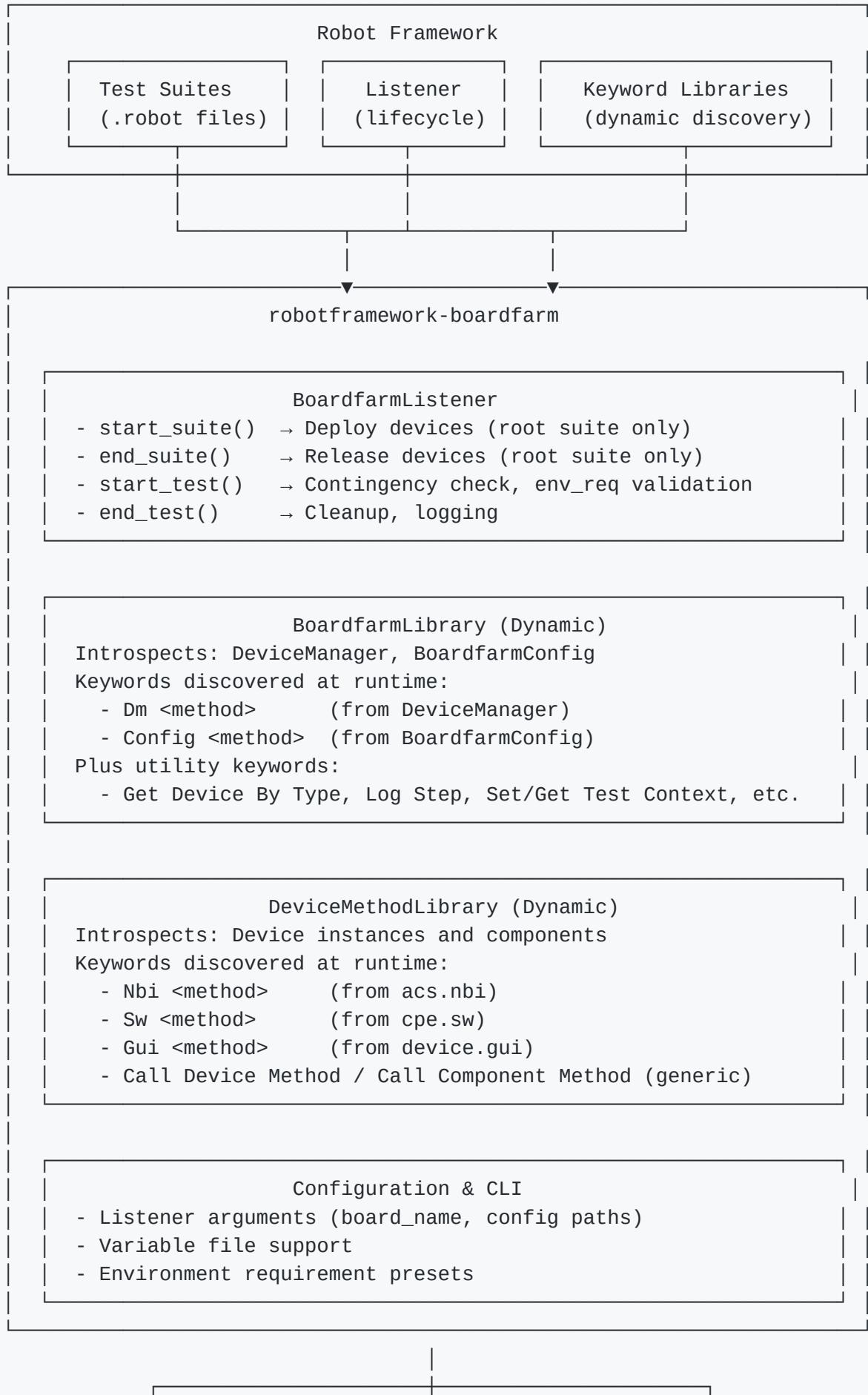
Robot Framework Extension Points

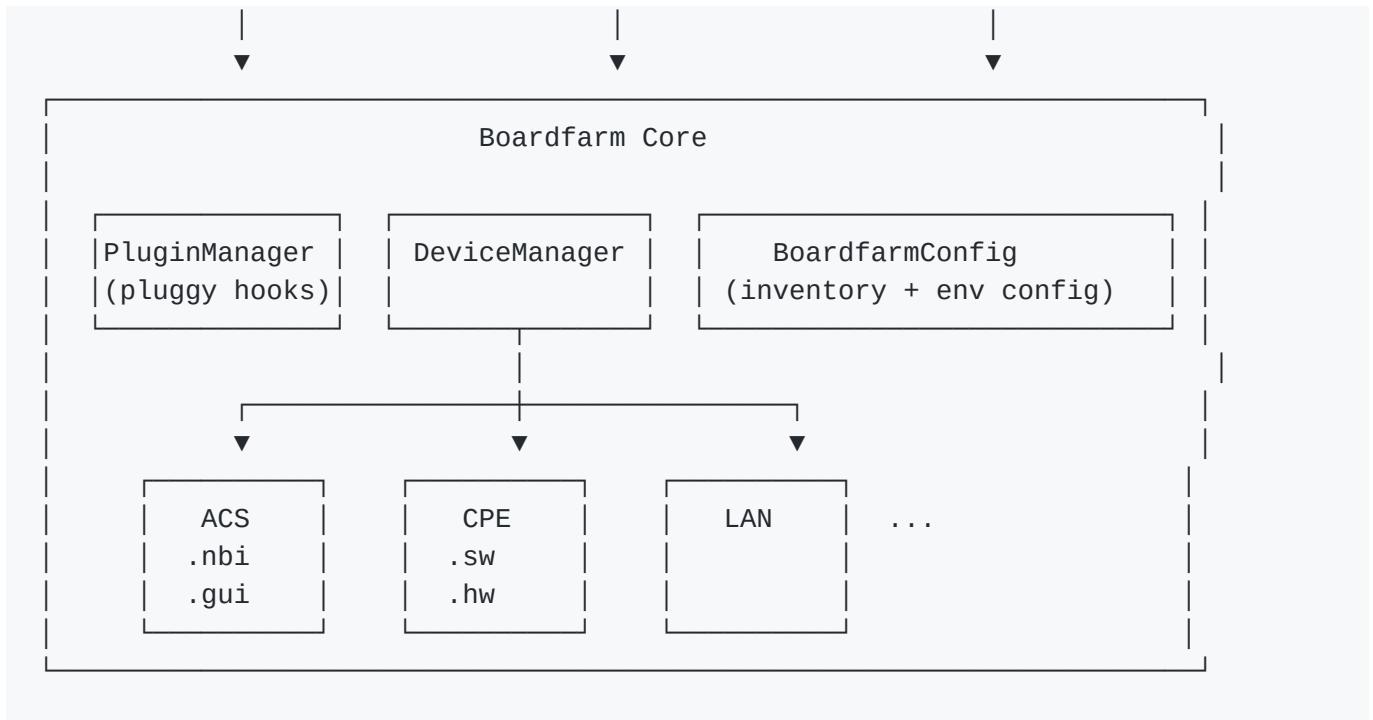
Robot Framework provides several extension mechanisms:

1. **Listener Interface** - For test lifecycle hooks (similar to pytest hooks)
 2. **Library API** - For creating keyword libraries
 3. **Visitor API** - For modifying test data
 4. **Pre-run Modifiers** - For filtering/modifying tests before execution
 5. **Result Visitors** - For processing execution results
-

Architecture Design

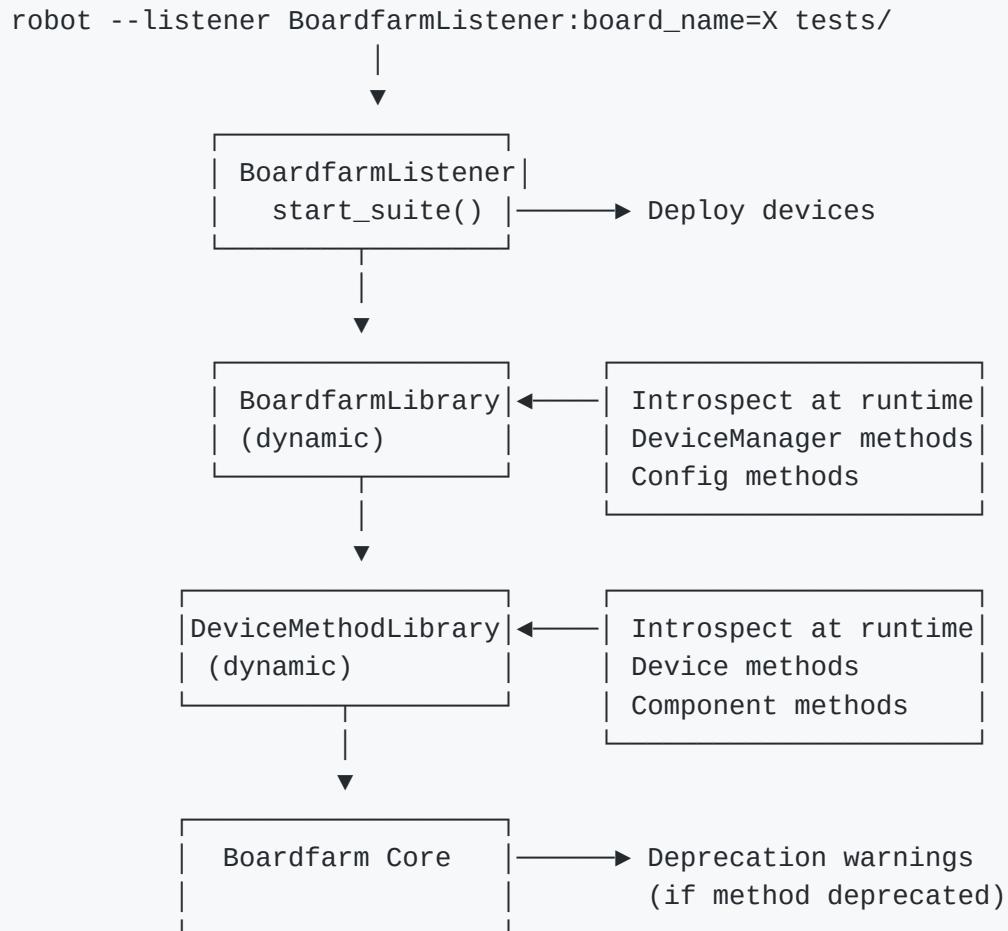
Component Overview





Data Flow

Test Execution:



Key Design Decisions

1. Listener-Based Lifecycle Management

- Use Robot Framework's Listener API (v3) for suite/test lifecycle
- Deploy devices at suite start, release at suite end
- Validate environment requirements at test start

2. Dynamic Discovery Architecture

- **BoardfarmLibrary**: Introspects DeviceManager and BoardfarmConfig
- **DeviceMethodLibrary**: Introspects device instances and components
- Both libraries adapt automatically when Boardfarm's API evolves

3. Future-Proof API Access

- Both libraries use introspection to discover methods at runtime
- Automatically adapts to new methods in future Boardfarm releases
- No code changes needed when APIs evolve
- Documentation auto-generated from method docstrings

4. Configuration via Variables

- Board name, config paths via `--variable` or variable files
- Support for `--listener` arguments
- Compatible with existing Boardfarm config format

5. Tag-Based Environment Requirements

- Use Robot Framework tags instead of pytest markers
- `[Tags] env_req:dual_stack` style tagging
- Pre-run modifier for filtering incompatible tests

Keyword Discovery Strategy

Problem Statement

Boardfarm exposes methods through multiple layers:

- `DeviceManager` : Device access and management

- BoardfarmConfig : Configuration access
- Device classes (CPE, ACS, LAN): Device-specific operations
- Device components (nbi, gui, sw, hw): Nested functionality

All of these interfaces can evolve with Boardfarm releases.

Solution: Dynamic Discovery at All Levels

| |
|--|
| BoardfarmLibrary Scope: Testbed infrastructure Source: DeviceManager, BoardfarmConfig instances Discovery: Introspects public methods at runtime Keywords: Dm <method>, Config <method>, plus utilities |
|--|

| |
|---|
| DeviceMethodLibrary Scope: Device instances Source: Device classes and components (nbi, gui, sw, hw) Discovery: Introspects device and component methods at runtime Keywords: <Component> <method>, Call Device Method |
|---|

Why Two Libraries?

| Library | What it introspects | Keyword prefix |
|---------------------|---------------------|-----------------------|
| BoardfarmLibrary | DeviceManager | Dm |
| BoardfarmLibrary | BoardfarmConfig | Config |
| DeviceMethodLibrary | Device components | Nbi , Sw , Gui , etc. |

Both use the same discovery pattern, just on different objects.

BoardfarmLibrary

Introspects testbed infrastructure:

```

*** Settings ***
Library BoardfarmLibrary

*** Test Cases ***
Test Testbed Infrastructure
    # DeviceManager methods (prefix: Dm)
    ${cpe}=    Dm Get Device By Type    ${CPE_CLASS}
    ${devices}=   Dm Get Devices By Type   ${LAN_CLASS}

    # BoardfarmConfig methods (prefix: Config)
    ${mode}=    Config Get Prov Mode
    ${sku}=    Config Get Board Sku
    ${device_config}=  Config Get Device Config    board

    # Utility keywords (Robot Framework specific)
    ${dm}=    Get Device Manager
    ${cpe}=    Get Device By Type    CPE      # Resolves type from string
    Log Step    Test step message
    Set Test Context    key    value

```

DeviceMethodLibrary

Introspects device instances and components:

```

*** Settings ***
Library DeviceMethodLibrary    device_type=ACS    WITH NAME    ACS
Library DeviceMethodLibrary    device_type=CPE    WITH NAME    CPE

*** Test Cases ***
Test Device Methods
    # ACS NBI methods (auto-discovered from acs.nbi)
    ${result}=    ACS.Nbi GPV    Device.DeviceInfo.Manufacturer
    ${status}=    ACS.Nbi SPV    {"Device.WiFi.SSID.1.SSID": "NewName"}
    ACS.Nbi Reboot    command_key=test

    # CPE SW methods (auto-discovered from cpe.sw)
    ${uptime}=    CPE.Sw Get Seconds Uptime
    ${online}=    CPE.Sw Is Online
    CPE.Sw Reset

    # Generic fallback (works with any method)
    ${result}=    ACS.Call Device Method    ACS    nbi.GPN    Device.
next_level=${TRUE}

```

How Dynamic Discovery Works

```
# Same pattern used by both libraries:

def get_keyword_names(self):
    keywords = []

    # Discover methods from target objects
    for name in dir(target_object):
        if callable(getattr(target_object, name)) and not name.startswith("_"):
            keywords.append(f"{prefix} {self._to_keyword_name(name)}")

    return keywords

def run_keyword(self, name, args, kwargs):
    # Parse keyword name to find target method
    prefix, method_name = self._parse_keyword(name)
    target = self._get_target(prefix)
    method = getattr(target, method_name)
    return method(*args, **kwargs)
```

Handling Boardfarm Updates

When Boardfarm adds new methods to ANY component:

| Component | Example | Result |
|-----------------|--------------------------|----------------------------------|
| DeviceManager | dm.new_query_method() | Dm New Query Method available |
| BoardfarmConfig | config.get_new_setting() | Config Get New Setting available |
| ACS NBI | acs.nbi.NewOperation() | ACS.Nbi New Operation available |
| CPE SW | cpe.sw.new_feature() | CPE.Sw New Feature available |

No code changes to robotframework-boardfarm required!

Deprecation Handling

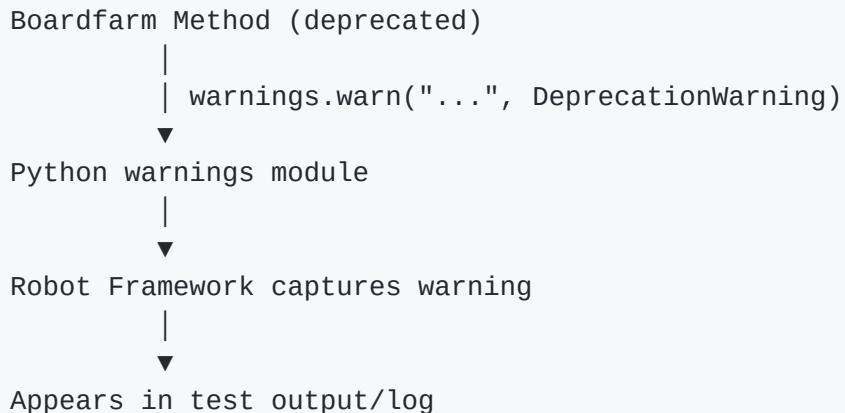
Design Principle: Single Source of Truth

Deprecation warnings are handled **by Boardfarm itself**, not by the integration layer.

Why?

- Avoids duplicate deprecation logic in multiple places
- Ensures consistent warnings for all Boardfarm users (pytest, Robot Framework, direct API)
- When Boardfarm updates deprecations, all integrations automatically get the change
- Follows the principle: handle concerns at the source

How It Works



When a Boardfarm method is deprecated:

```
# In Boardfarm (boardfarm3/templates/acs/acs_nbi.py)
def GetParameterValues(self, ...):
    """Deprecated: Use GPV instead."""
    warnings.warn(
        "GetParameterValues is deprecated, use GPV instead",
        DeprecationWarning,
        stacklevel=2
    )
    return self.GPV(...)
```

The warning automatically appears in Robot Framework output:

```
WARN: DeprecationWarning: GetParameterValues is deprecated, use GPV instead
```

Boardfarm Responsibilities

Boardfarm should:

1. **Emit warnings** via Python's `warnings` module when deprecated methods are called

2. Document deprecations in docstrings (picked up by dynamic documentation)
3. Optionally maintain a registry for programmatic queries

```
# Recommended pattern in Boardfarm
import warnings
from functools import wraps

def deprecated(message: str, replacement: str | None = None):
    """Decorator to mark methods as deprecated."""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            warn_msg = f"{func.__name__} is deprecated. {message}"
            if replacement:
                warn_msg += f" Use {replacement} instead."
            warnings.warn(warn_msg, DeprecationWarning, stacklevel=2)
            return func(*args, **kwargs)
        wrapper.__deprecated__ = message
        return wrapper
    return decorator

# Usage in Boardfarm
@deprecated("Renamed for consistency", replacement="GPV")
def GetParameterValues(self, ...):
    return self.GPV(...)
```

robotframework-boardfarm Role

The integration layer simply:

- **Passes through** whatever warnings Boardfarm emits
- **Does not** implement its own deprecation detection
- **Does not** maintain separate deprecation state

This keeps the integration simple and ensures users get the same experience regardless of how they use Boardfarm.

Implementation Phases

Phase 1: Project Foundation (1-2 days)

Goal: Establish project structure, dependencies, and basic infrastructure.

Deliverables:

- Project structure with `pyproject.toml`
- Basic package layout
- Development tooling configuration (ruff, mypy, pytest)
- README with overview and goals
- CI/CD configuration (GitHub Actions)

Files to Create:

```
robotframework-boardfarm/
├── pyproject.toml
├── README.md
├── LICENSE
├── .gitignore
├── .pre-commit-config.yaml
├── .flake8
├── .pylintrc
├── noxfile.py
└── robotframework_boardfarm/
    ├── __init__.py
    ├── py.typed
    └── version.py
└── tests/
    └── __init__.py
```

Phase 2: Listener Implementation (2-3 days)

Goal: Implement the core lifecycle management listener.

Deliverables:

- `BoardfarmListener` class implementing Listener API v3
- Suite-level device deployment/teardown
- Test-level contingency checks
- Logging integration
- Error handling and recovery

Key Methods:

```
class BoardfarmListener:  
    ROBOT_LISTENER_API_VERSION = 3  
  
    def start_suite(self, data, result):  
        """Deploy boardfarm devices at suite start."""  
  
    def end_suite(self, data, result):  
        """Release boardfarm devices at suite end."""  
  
    def start_test(self, data, result):  
        """Validate env requirements and run contingency checks."""  
  
    def end_test(self, data, result):  
        """Cleanup and capture logs."""
```

Phase 3: Keyword Libraries (2-3 days)

Goal: Create dynamic keyword libraries for testbed and device access.

Deliverables:

- `BoardfarmLibrary` - Dynamic library introspecting DeviceManager/Config
- `DeviceMethodLibrary` - Dynamic library introspecting device instances
- `DeprecationHandler` - Deprecation detection and warning system
- Documentation auto-generated from introspected methods

BoardfarmLibrary Keywords (dynamically discovered + utilities):

```

# Discovered from DeviceManager (prefix: Dm)
Dm Get Device By Type    ${device_class}
Dm Get Devices By Type   ${device_class}

# Discovered from BoardfarmConfig (prefix: Config)
Config Get Prov Mode
Config Get Board Sku
Config Get Device Config  ${device_name}

# Utility keywords (Robot Framework specific)
Get Device By Type    ${device_type}      # Resolves type from string
Get Device Manager
Get Boardfarm Config
Log Step    ${message}
Set Test Context  ${key}    ${value}
Get Test Context  ${key}
Require Environment ${requirement}

```

DeviceMethodLibrary Keywords (dynamically discovered):

```

# From ACS device (device_type=ACS)
Nbi GPV    ${parameters}
Nbi SPV    ${param_values}
Nbi Reboot
Gui Login  ${username}  ${password}

# From CPE device (device_type=CPE)
Sw Get Seconds Uptime
Sw Is Online
Sw Reset

# Generic fallback
Call Device Method  ${device_type}  ${method_path}  @{args}
Call Component Method ${device_type}  ${component}  ${method}  @{args}

```

Phase 4: Configuration & CLI Integration (1-2 days)

Goal: Handle configuration loading and command-line arguments.

Deliverables:

- Variable file for Robot Framework
- Argument forwarding to Boardfarm hooks
- Config validation and merging

- Pre-run modifier for test filtering

Usage Pattern:

```
robot --listener robotframework_boardfarm.BoardfarmListener \
--variable BOARD_NAME:prplos-docker-1 \
--variable ENV_CONFIG:./bf_config/boardfarm_env.json \
--variable INVENTORY_CONFIG:./bf_config/boardfarm_config.json \
--variablefile robotframework_boardfarm/variables.py \
tests/
```

Phase 5: Environment Requirement Support (1-2 days)

Goal: Implement tag-based environment filtering (equivalent to `@pytest.mark.env_req`).

Deliverables:

- Pre-run modifier for `env_req` tag parsing
- Environment matching logic (port from pytest-boardfarm)
- Test skipping for incompatible environments
- Documentation on tag syntax

Tag Syntax:

```
*** Test Cases ***
Test Dual Stack Mode
[Tags]    env_req:{"environment_def":{"board":{"eRouter_Provisioning_mode": "dual"}}}
# Test implementation
```

Phase 6: Reporting Integration (1-2 days)

Goal: Enhance Robot Framework reports with Boardfarm information.

Deliverables:

- Result visitor for report enhancement
- Deployment status in report
- Device information in report
- Console log attachments
- Screenshot support

Phase 7: Testing & Documentation (2-3 days)

Goal: Comprehensive testing and documentation.

Deliverables:

- Unit tests for all components
- Integration tests with mock devices
- Example test suites
- API documentation
- User guide
- Migration guide (from pytest to Robot Framework)

Phase 8: Validation & Refinement (2-3 days)

Goal: Test against real testbed and refine based on feedback.

Deliverables:

- Execute existing BDD scenarios ported to Robot Framework
 - Performance benchmarking
 - Error message improvements
 - Edge case handling
 - Final documentation polish
-

Detailed Component Specifications

1. BoardfarmListener

```

"""Robot Framework listener for Boardfarm integration."""

from __future__ import annotations

import asyncio
import logging
from argparse import Namespace
from typing import TYPE_CHECKING, Any

from boardfarm3.lib.boardfarm_config import BoardfarmConfig, get_json
from boardfarm3.lib.device_manager import DeviceManager
from boardfarm3.main import get_plugin_manager

if TYPE_CHECKING:
    from robot.running import TestSuite, TestCase
    from robot.result import TestSuite as ResultSuite, TestCase as ResultCase


class BoardfarmListener:
    """Robot Framework listener for Boardfarm device lifecycle management."""

    ROBOT_LISTENER_API_VERSION = 3
    ROBOT_LIBRARY_SCOPE = "GLOBAL"

    def __init__(
        self,
        board_name: str | None = None,
        env_config: str | None = None,
        inventory_config: str | None = None,
        skip_boot: bool = False,
        skip_contingency_checks: bool = False,
        save_console_logs: str | None = None,
    ) -> None:
        """Initialize boardfarm listener.

Args:
    board_name: Name of the board to use
    env_config: Path to environment JSON config
    inventory_config: Path to inventory JSON config
    skip_boot: Skip device booting if True
    skip_contingency_checks: Skip contingency checks if True
    save_console_logs: Path to save console logs
"""

        self._board_name = board_name
        self._env_config_path = env_config
        self._inventory_config_path = inventory_config
        self._skip_boot = skip_boot

```

```

self._skip_contingency_checks = skip_contingency_checks
self._save_console_logs = save_console_logs

self._plugin_manager = get_plugin_manager()
self._device_manager: DeviceManager | None = None
self._boardfarm_config: BoardfarmConfig | None = None
self._deployment_data: dict[str, Any] = {}
self._logger = logging.getLogger("boardfarm.robotframework")

@property
def device_manager(self) -> DeviceManager:
    """Return device manager instance."""
    if self._device_manager is None:
        raise RuntimeError("Device manager not initialized")
    return self._device_manager

@property
def boardfarm_config(self) -> BoardfarmConfig:
    """Return boardfarm config instance."""
    if self._boardfarm_config is None:
        raise RuntimeError("Boardfarm config not initialized")
    return self._boardfarm_config

def start_suite(self, data: TestSuite, result: ResultSuite) -> None:
    """Deploy boardfarm devices at suite start.

    Only deploys for the root suite (top-level).
    """
    if data.parent is None: # Root suite only
        self._deploy_devices()

def end_suite(self, data: TestSuite, result: ResultSuite) -> None:
    """Release boardfarm devices at suite end.

    Only releases for the root suite (top-level).
    """
    if data.parent is None: # Root suite only
        self._release_devices()

def start_test(self, data: TestCase, result: ResultCase) -> None:
    """Validate environment requirements before test execution."""
    if not self._skip_contingency_checks:
        env_req = self._parse_env_req_tags(data.tags)
        if env_req:
            self._validate_env_requirement(env_req)

def end_test(self, data: TestCase, result: ResultCase) -> None:
    """Cleanup after test execution."""

```

```
pass # Future: capture logs, cleanup context

def _deploy_devices(self) -> None:
    """Deploy boardfarm devices."""
    # Implementation follows pytest-boardfarm pattern
    ...

def _release_devices(self) -> None:
    """Release boardfarm devices."""
    # Implementation follows pytest-boardfarm pattern
    ...

def _parse_env_req_tags(self, tags: list[str]) -> dict[str, Any] | None:
    """Parse env_req tags from test tags."""
    ...

def _validate_env_requirement(self, env_req: dict[str, Any]) -> None:
    """Validate environment meets test requirements."""
    ...
```

2. BoardfarmLibrary (Dynamic)

```

"""Robot Framework keyword library for Boardfarm testbed management.

Uses dynamic discovery to introspect DeviceManager and BoardfarmConfig,
automatically exposing their methods as keywords.

Note: Deprecation warnings are handled by Boardfarm itself, not this library.
"""

from __future__ import annotations

import inspect
from typing import TYPE_CHECKING, Any

from robot.api import logger
from robot.api.deco import keyword, library

if TYPE_CHECKING:
    from boardfarm3.lib.boardfarm_config import BoardfarmConfig
    from boardfarm3.lib.device_manager import DeviceManager


@library(scope="GLOBAL", version="0.1.0")
class BoardfarmLibrary:
    """Dynamic library for Boardfarm testbed management.

    Introspects DeviceManager and BoardfarmConfig to discover keywords.
    Also provides utility keywords for Robot Framework specific needs.

    Example:
    | *** Settings ***
    | Library      BoardfarmLibrary
    |
    | *** Test Cases ***
    | Test Dynamic Discovery
    |     # Discovered from BoardfarmConfig
    |     ${mode}=  Config Get Prov Mode
    |     # Utility keyword
    |     ${cpe}=  Get Device By Type  CPE
    """

    # Components to introspect for dynamic keywords
    _INTROSPECT_COMPONENTS = {
        "device_manager": "Dm",      # DeviceManager -> "Dm <Method>"
        "boardfarm_config": "Config", # BoardfarmConfig -> "Config <Method>"
    }

    def __init__(self) -> None:

```

```

self._context: dict[str, Any] = {}
self._keyword_cache: dict[str, tuple[str, str, Any]] = {}

# === Dynamic Library Protocol ===

def get_keyword_names(self) -> list[str]:
    """Discover keywords from DeviceManager and BoardfarmConfig."""
    keywords = []

    # Add utility keywords (decorated with @keyword)
    for name in dir(self):
        method = getattr(self, name, None)
        if callable(method) and hasattr(method, "robot_name"):
            keywords.append(method.robot_name)

    # Add dynamic keywords from testbed components
    try:
        listener = get_listener()
        for attr_name, prefix in self._INTROSPECT_COMPONENTS.items():
            obj = getattr(listener, attr_name, None)
            if obj is not None:
                keywords.extend(self._discover_keywords(obj, prefix))
    except Exception:
        pass # Listener not ready

    return keywords

def run_keyword(self, name: str, args: list, kwargs: dict) -> Any:
    """Execute a keyword. Deprecation warnings come from Boardfarm."""
    return self._execute_keyword(name, args, kwargs)

# === Utility Keywords (Robot Framework specific) ===

@keyword("Get Device By Type")
def get_device_by_type(self, device_type: str):
    """Get device by type name (resolves string to class)."""
    ...

@keyword("Log Step")
def log_step(self, message: str) -> None:
    """Log a test step message."""
    logger.info(f"[STEP] {message}", html=False)

@keyword("Set Test Context")
def set_test_context(self, key: str, value: Any) -> None:
    """Store a value in the test context."""
    self._context[key] = value

```

```
@keyword("Require Environment")
def require_environment(self, requirement: dict[str, Any]) -> None:
    """Assert environment meets requirement or skip test."""
    from robot.api import SkipExecution
    if not is_env_matching(requirement, config.env_config):
        raise SkipExecution("Environment requirement not met")
```

3. DeviceMethodLibrary (Dynamic)

```
"""Dynamic library that exposes device methods as keywords.
```

```
Introspects device instances and their components (nbi, gui, sw, hw)  
to automatically discover and expose methods as Robot Framework keywords.
```

```
Note: Deprecation warnings are handled by Boardfarm itself, not this library.
```

```
from __future__ import annotations  
  
import inspect  
from typing import Any
```

```
class DeviceMethodLibrary:  
    """Dynamic library for device method access.
```

```
Introspects device instances to discover methods at runtime.  
Supports nested components like acs.nbi, cpe.sw, cpe.hw.
```

```
Example:
```

```
| *** Settings ***
| Library     DeviceMethodLibrary      device_type=ACS      WITH NAME      ACS
| Library     DeviceMethodLibrary      device_type=CPE      WITH NAME      CPE
|
| *** Test Cases ***
| Test Device Methods
|     ${result}=    ACS.Nbi GPV      Device.DeviceInfo.
|     ${uptime}=    CPE.Sw Get Seconds Uptime
"""

```

```
ROBOT_LIBRARY_SCOPE = "GLOBAL"
```

```
def __init__(self, device_type: str | None = None) -> None:
    self._device_type = device_type
    self._keyword_cache: dict[str, Any] = {}

def get_keyword_names(self) -> list[str]:
    """Discover keywords from device and component methods."""
    keywords = ["Call Device Method", "Call Component Method"]

    if self._device_type:
        device = self._get_device()
        # Discover direct device methods
        keywords.extend(self._discover_methods(device, ""))
        # Discover component methods (nbi, gui, sw, hw)
        for comp_name in ["nbi", "gui", "sw", "hw", "console"]:
```

```

        if hasattr(device, comp_name):
            component = getattr(device, comp_name)
            prefix = comp_name.title()
            keywords.extend(self._discover_methods(component, prefix))

    return keywords

def run_keyword(self, name: str, args: list, kwargs: dict) -> Any:
    """Execute keyword. Deprecation warnings come from Boardfarm."""
    if name == "Call Device Method":
        return self._call_device_method(*args, **kwargs)
    elif name == "Call Component Method":
        return self._call_component_method(*args, **kwargs)
    else:
        return self._run_discovered_keyword(name, args, kwargs)

def _discover_methods(self, obj: Any, prefix: str) -> list[str]:
    """Discover callable methods from an object."""
    keywords = []
    for name in dir(obj):
        if name.startswith("_"):
            continue
        method = getattr(obj, name, None)
        if callable(method) and not isinstance(method, type):
            kw_name = f"{prefix} {self._to_keyword_name(name)}".strip()
            keywords.append(kw_name)
            self._keyword_cache[kw_name] = method
    return keywords

def _call_device_method(self, device_type: str, method_path: str, *args,
                       **kwargs):
    """Generic keyword to call any device method."""
    ...

def _call_component_method(self, device_type: str, component: str, method: str,
                           *args, **kwargs):
    """Generic keyword to call any component method."""
    ...

```

4. Variable File

```

"""Robot Framework variable file for Boardfarm configuration."""

import os
from typing import Any

def get_variables(
    board_name: str | None = None,
    env_config: str | None = None,
    inventory_config: str | None = None,
) -> dict[str, Any]:
    """Return variables for Robot Framework.

    Variables can be overridden via command line:
        robot --variable BOARD_NAME:my-board tests/
    """

    Args:
        board_name: Board name (can be set via BOARD_NAME env var)
        env_config: Environment config path
        inventory_config: Inventory config path

    Returns:
        Dictionary of variables for Robot Framework.
    """

    return {
        "BOARD_NAME": board_name or os.environ.get("BOARD_NAME", ""),
        "ENV_CONFIG": env_config or os.environ.get("ENV_CONFIG", ""),
        "INVENTORY_CONFIG": inventory_config or os.environ.get("INVENTORY_CONFIG",
        ""),
        "SKIP_BOOT": os.environ.get("SKIP_BOOT", "false").lower() == "true",
        "SKIP_CONTINGENCY_CHECKS": os.environ.get(
            "SKIP_CONTINGENCY_CHECKS", "false"
        ).lower() == "true",
    }

```

Usage Examples

Basic Test Suite

```

*** Settings ***
Library          BoardfarmLibrary
Suite Setup      Log    Starting Boardfarm tests
Suite Teardown   Log    Completed Boardfarm tests

*** Test Cases ***
Test Device Connection
[Documentation] Verify device connectivity
${cpe}=  Get Device By Type CPE
Log    Connected to CPE: ${cpe}

Test Provisioning Mode
[Documentation] Verify provisioning mode
${mode}=  Get Provisioning Mode
Log    Provisioning mode: ${mode}
Should Be Equal ${mode} dual

Test With Environment Requirement
[Documentation] Test requiring dual stack
[Tags] env_req:dual_stack
Require Environment ${ENV_REQ_DUAL_STACK}
Log Step Environment validated
# Test implementation

```

Command Line Execution

```

# Basic execution
robot --listener robotframework_boardfarm.BoardfarmListener:board_name=prplos-docker-1:env_config=./bf_config/boardfarm_env.json:inventory_config=./bf_config/boardfarm_config.json \
tests/

# Using variables
robot --variable BOARD_NAME:prplos-docker-1 \
--variable ENV_CONFIG:./bf_config/boardfarm_env.json \
--variable INVENTORY_CONFIG:./bf_config/boardfarm_config.json \
--listener robotframework_boardfarm.BoardfarmListener \
tests/

# Skip boot for faster iteration
robot --variable SKIP_BOOT:true \
--listener robotframework_boardfarm.BoardfarmListener \
tests/

```

Dependencies

```
[project]
dependencies = [
    "robotframework>=6.0",
    "boardfarm3>=1.0.0",
]

[project.optional-dependencies]
dev = [
    "pytest",
    "pytest-cov",
    "ruff",
    "mypy",
    "pre-commit",
]
```

Testing Strategy

Unit Tests

- Test listener lifecycle methods with mocked Boardfarm
- Test keyword library with mocked device manager
- Test configuration loading and merging
- Test environment matching logic

Integration Tests

- Test with mock devices (no real hardware)
- Verify listener/library interaction
- Test error handling and recovery

System Tests

- Execute against real testbed
 - Port existing BDD scenarios
 - Verify report generation
-

Timeline Estimate

| Phase | Duration | Dependencies |
|---------------------------|----------|--------------|
| Phase 1: Foundation | 1-2 days | None |
| Phase 2: Listener | 2-3 days | Phase 1 |
| Phase 3: Keywords | 2-3 days | Phase 2 |
| Phase 4: Configuration | 1-2 days | Phase 2 |
| Phase 5: Env Requirements | 1-2 days | Phase 3, 4 |
| Phase 6: Reporting | 1-2 days | Phase 3 |
| Phase 7: Testing/Docs | 2-3 days | All |
| Phase 8: Validation | 2-3 days | Phase 7 |

Total Estimated Duration: 12-20 days

Risks and Mitigations

| Risk | Impact | Mitigation |
|---------------------------------------|--------|---|
| Async operations in Robot Framework | Medium | Use <code>asyncio.run()</code> wrapper, test thoroughly |
| Device state persistence across tests | High | Clear state in listener hooks, document cleanup |
| Report customization limitations | Low | Use result visitors, custom log messages |
| Performance overhead | Medium | Lazy initialization, caching |

Success Criteria

1. Execute existing Boardfarm test scenarios in Robot Framework
2. Device deployment and teardown works reliably
3. Environment requirements filter tests correctly

4. Reports include Boardfarm deployment information
 5. Documentation enables self-service adoption
 6. Performance comparable to pytest-boardfarm
-

Next Steps

1. Create project structure (Phase 1)
 2. Implement core listener (Phase 2)
 3. Iterate on keyword library based on usage patterns
 4. Validate with real testbed scenarios
-

Document Version: 1.0

Last Updated: January 17, 2025