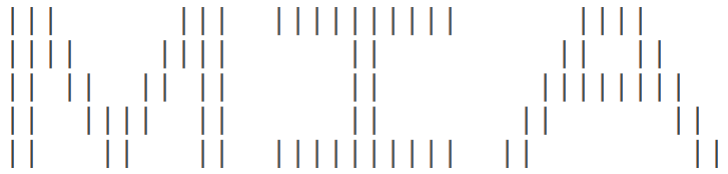

Multiple Integrated Applications (MIA) Manual & Programming Documentation

Developer: Antonius Torode

Version – 2.000
Latest update: May 23, 2025

The Original Maintainer of this work is: Antonius Torode.

The Current Maintainer of this work is: Antonius Torode.

This document is designed for the sole purpose of providing documentation for Multiple Integrated Applications (MIA), a program written for and created for personal use. Throughout this document, "MIA" will be used as a reference to "Multiple Integrated Applications." This acronym is solely designed for use in conjunction with the program and was originally created by the creator of this document. MIA is designed to be used for multiple purposes based on the continual addition of functionality. It can be adapted to work in other situations and for alternate purposes.

This document is continuously under development.
Most Current Revision Date: May 23, 2025

Torode, A.
MIA Manual.
2021.
Includes Source Code and References

Contents

This page intentionally left blank.
(Yes, this is a contradiction)

Chapter 1

Multiple Integrated Applications (MIA)

1.1 Introduction

MIA is designed to be a collection of scripts, tools, programs, and commands that have been created in the past and may be useful in the future. It's original idea was a place for the original author to combine all of his previous applications and codes into one location that can be compiled cross platform. MIA is written in C++ but will contain codes that were originally designed in C#, Java, Python, and others. MIA is created for the authors personal use but may be used by others if a need or desire arises under the terms of Antonius' General Purpose License (AGPL).

The MIA acronym was created by the original author for the sole purpose of this application. The design of MIA is a terminal prompt that accepts commands. There are no plans to convert MIA into a GUI application as there is currently no need; however, some elements may be programmed in that produce a GUI window for certain uses such as graphs. The MIA manual is designed to be an explanation of what MIA contains as well as a guide of how to utilize the MIA program to it's fullest.

As MIA is continually under development, this document is also. Due to this, it may fall behind and become slightly outdated as I implement and test new features into MIA. I will attempt to keep this document up to date with all of the features MIA contains but I can only do so if time permits.

Chapter 2

MIA Development

2.1 Application Framework

Overview

The application framework provides a minimal structure for C++ applications that follow a consistent life cycle. It enforces a standard interface consisting of initialization and execution phases, and offers utilities to streamline application entry point definition. The design leverages C++20 concepts and templates for compile-time enforcement of interface contracts.

Life Cycle Model

Applications using this framework must implement the following life cycle methods:

- `void initialize(int argc, char** argv)`
  Performs startup logic, such as argument parsing and resource allocation.

- `int run()`
  Contains the main execution logic of the application. The return value is propagated as the process exit code.

Interface Enforcement

The framework uses a C++20 concept, `AppInterface`, to statically constrain application types. This ensures that any type passed to the launcher function conforms to the expected interface, eliminating runtime errors due to missing methods.

Application Launch

Applications are launched using the `runApp` template function:

- Accepts any type satisfying `AppInterface`.

- Calls `initialize` followed by `run`, in that order.

- Returns the result of `run` as the application's exit code.

Entry Point Definition

To reduce boilerplate and unify application entry points, the framework provides a macro:

- `MIA_MAIN(AppClass)` defines a `main()` function that delegates to `runApp<AppClass>`.

- This enables consistent startup across applications while preserving type safety and clarity.

Usage Guidelines

1. Define an application class that implements the required `initialize` and `run` methods.
2. Use `MIA_MAIN(YourAppClass);` in a translation unit to define the application entry point.
3. Avoid manual `main()` definitions to preserve uniform lifecycle management.

Benefits

- Promotes a consistent lifecycle across applications.
- Enables interface enforcement at compile time.
- Minimizes boilerplate in entry point logic.
- Facilitates cleaner and more maintainable application architecture.

2.2 Base Application Class: `MIAApplication`

Overview

The base MIA Application (see `MIAApplication.hpp`) serves as the foundational base class for applications built using the framework. It defines a standardized interface for application lifecycle methods and provides shared functionality for command-line argument parsing, particularly handling common flags such as `-v` (verbose) and `-h` (help).

Purpose

This class is intended to be subclassed by specific applications. It provides default behavior for argument parsing and flag handling, while enforcing the implementation of core logic via a pure virtual `run()` method.

Initialization Interface

- `virtual void initialize(int argc, char* argv[])`
  Handles initial setup, including parsing common command-line arguments. Derived classes should override this method to extend parsing logic but should still invoke the base implementation to preserve flag handling.

Execution Interface

- `virtual int run() = 0`
  Pure virtual function that must be implemented by all derived classes. It defines the main operational logic of the application and must return an integer exit code.

Flag Handling

- Verbose mode is handled internally and can be queried via `getVerboseMode()`.
- Help flag status is stored and can be utilized to trigger usage messages.
- `printHelp()` provides a virtual method to emit shared help information; it can be extended or overridden by subclasses.

Protected Members

- `bool verboseMode`
  Indicates whether verbose output was requested.
- `bool helpRequested`
  Set to `true` if the user requested help via the command-line.

Usage Guidelines

1. Inherit from `MIAApplication`.
2. Override `initialize()` to add application-specific argument parsing; call the base method to retain common flag handling.
3. Implement the `run()` method with the application's main logic.
4. Use `getVerboseMode()` to conditionally enable verbose output.

Benefits

- Standardizes lifecycle structure across applications.
- Centralizes command-line flag logic.
- Reduces redundancy in application setup.
- Encourages consistent help and verbose interfaces.

2.3 Command Option System

The command option system provides structured parsing and management of command-line arguments. It is composed of two main components: the `CommandOption` class and the `command_parser` namespace. Together, they offer a consistent and type-safe interface for defining, parsing, and validating command-line options.

2.3.1 CommandOption Class

The `CommandOption` class encapsulates metadata and behavior for individual command-line arguments. It supports a variety of data types and provides a standardized help display format.

Key Features

- **Supported Types:** Defined by the `commandOptionType` enum:
  - `boolOption`
  - `intOption`
  - `doubleOption`
  - `stringOption`
- **Constructor:** Accepts short/long argument forms, a description, data type, and a `required` flag.
- **Help Output:** `getHelp()` returns a formatted string of the form:

```
-s, --long        Description
```

- **Value Parsing:** The `getOptionVal<Type>` method retrieves the typed value from the command-line using appropriate dispatch.
- **Error Handling:** Throws `MIAException` if a type mismatch occurs or parsing fails.

2.3.2 command_parser Namespace

This namespace implements the actual logic for extracting typed values from `argc`/`argv`. Each function accepts short and long option names, and a reference to the variable where the parsed value should be stored.

Parsing Functions

- `void parseBoolFlag(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, bool& outValue);`
- `void parseIntOption(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, int& outValue, bool required = false);`
- `void parseDoubleOption(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, double& outValue, bool required = false);`
- `void parseStringOption(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, std::string& outValue, bool required = false);`

Behavior

Each parser:

- Validates the presence of the option.
- Converts the argument to the correct type.
- Throws a `MIAException` on error, such as type mismatch or missing required value.

2.3.3 Integration

The `CommandOption` class acts as a high-level interface, while `command_parser` performs the low-level parsing. Together, they ensure robustness, type safety, and consistent error reporting across the command-line interface.

2.4 Error Handling Framework

Overview

The MIA error handling system provides a structured and consistent mechanism for reporting, categorizing, and propagating errors throughout MIA applications. It is centered around a custom exception type (`MIAException`) and an extensible set of error codes defined via the `ErrorCode` enumeration.

Error Codes

The `ErrorCode` enumeration defines a comprehensive list of standardized error values used throughout the MIA framework. These include:

- System-aligned codes (e.g., `Access_denied = 5`)
- Application-specific codes starting at 31415 (i.e., `int`($\pi \times 10000$)), e.g., `Feature_In_Dev`
- Custom fatal and configuration errors (e.g., `FATAL_File_Not_Found`, `Config_File_Not_Set`)

Each error code is associated with a human-readable description defined in `ErrorDescriptions.hpp`, accessible via:

```
const std::string& getErrorDescription(ErrorCode code);
```

MIAException Class

The primary mechanism for structured error propagation is the `MIAException` class:

- Inherits from `std::exception`.
- Encapsulates an `ErrorCode` and an optional detailed message.
- Supports integration with standard C++ `try/catch` error handling.

Interface Summary:

- `MIAException(ErrorCode, const std::string& details = "")`
  Constructs an exception with a specific code and optional extra context.
- `const char* what() const noexcept`
  Returns a descriptive error string combining code-based and contextual information.
- `ErrorCode getCode() const noexcept`
  Retrieves the error code associated with the exception.

Deprecated Legacy Functions

Several older functions exist but are marked deprecated in favor of `MIAException`:

- `returnError()`, `errorInfo()`, and `errorInfoRun()` are retained for backward compatibility but should be avoided.
- These are flagged with `[[deprecated]]` attributes.

Usage Guidelines

1. Throw `MIAException` in place of manual error returns.
2. Use `getErrorDescription()` to log or report known error codes.
3. Check exception codes via `getCode()` in catch blocks to take context-specific actions.

Benefits

- Unifies error handling across the application ecosystem.
- Enables richer debugging and logging through descriptive errors.
- Supports granular response logic based on typed error codes.

2.5 Global Constants and Paths

This section documents the centralized constants and path utilities used across the application. These components standardize access to configuration values, resource locations, and metadata such as the application version.

2.5.1 constants Namespace

The `constants` namespace provides globally accessible constant values compiled into the application.

Defined Constant

- `MIA_VERSION`: A `std::string` representing the version of the MIA application, injected from CMake via the `MIA_VERSION_VAL` macro.

2.5.2 paths Namespace

The `paths` namespace offers a centralized interface for accessing directory and file paths used by the system. These include installation-specific and repository-relative locations, as well as runtime-determined paths.

Static Path Constants

- `SYSTEM_CONFIG_FILE_DIR`, `SYSTEM_CONFIG_FILE`: Paths to the configuration directory and file for system installations.
- `REPO_CONFIG_FILE_DIR`, `REPO_CONFIG_FILE`: Paths used during development and testing.
- `SYSTEM_LOG_DIR`, `SYSTEM_LOG`: Default logging directory and file when installed.
- `REPO_LOG_DIR`, `REPO_LOG`: Logging paths for repository use.
- `INSTALL_LOCATION`: The root directory of the system installation.

Runtime Utilities

- `getExecutableDir()`: Returns the absolute directory of the running executable using platform-specific APIs.
- `isInstalled()`: Determines whether the application is running from the system-installed location by comparing the executable path with the install directory.
- `getDefaultConfigDirToUse()`: Selects the configuration directory based on execution context:
  - If installed: returns `SYSTEM_CONFIG_FILE_DIR`.
  - If a `resources` folder exists in the executable directory: returns that path.
  - Otherwise: returns `REPO_CONFIG_FILE_DIR`.

This design ensures portability and flexibility across different environments (development, testing, deployment), with consistent fallback logic and platform-agnostic path resolution.