





KissTelegram v5.0 User Manual

Table of Contents

1. [Introduction](#)
 2. [Migration from UniversalTelegramBot](#)
 3. [Key Features](#)
 4. [Technical Architecture](#)
 5. [Installation and Setup](#)
 6. [Message Persistence - Zero Loss](#)
 7. [Priority Management](#)
 8. [Field Power Management](#)
 9. [System Commands](#)
 10. [OTA Updates](#)
 11. [Practical Examples](#)
 12. [Troubleshooting](#)
-

Introduction

KissTelegram is an ESP32 framework for Telegram bots designed with one goal: **zero message loss**. Unlike other libraries, KissTelegram uses LittleFS as primary storage, not as backup. This ensures that:

-  WiFi failures → messages saved
-  ESP32 reboots → messages saved
-  Power outages → messages saved
-  OTA updates → messages saved

Why switch from UniversalTelegramBot?

- **Real persistence:** Messages survive any failure
 - **Priorities:** Critical messages jump the queue
 - **Power Management:** 6 energy-saving modes
 - **Safe OTA:** Dual-bank firmware updates with automatic rollback
 - **No dependencies:** Doesn't need ArduinoJson or external libraries
 - **No memory leaks:** Everything in char[], no Strings
 - **24/7 production:** Proper millis() overflow handling (49+ days)
 - **Intelligent WiFi:** Avoids race conditions and unnecessary work
-

Migration from UniversalTelegramBot

Code Comparison

UniversalTelegramBot (old):

```
cpp
#include <UniversalTelegramBot.h>

WiFiClientSecure client;
UniversalTelegramBot bot(BOT_TOKEN, client);

void setup() {
  WiFi.begin(SSID, PASSWORD);
  while (WiFi.status() != WL_CONNECTED) delay(500);
  client.setInsecure();
}

void loop() {
  // ❌ If WiFi fails here, message is lost
  bot.sendMessage(CHAT_ID, "Sensor: 25.3°C", "");
  delay(60000);
}
```

KissTelegram (new):

```
cpp
```

```
#include "KissTelegram.h"
#include "KissCredentials.h"

KissCredentials& creds = KissCredentials::getInstance();
KissTelegram* bot;

void setup() {
  creds.begin();
  WiFi.begin(creds.getWifiSSID(), creds.getWifiPassword());
  while (WiFi.status() != WL_CONNECTED) delay(500);

  bot = new KissTelegram(creds.getBotToken());
  bot->restoreFromStorage(); // ✅ Recovers pending messages
  bot->setWifiStable();
  bot->enable();
}

void loop() {
  bot->processQueue(); // ✅ Sends pending messages

  // ✅ If WiFi fails, message is saved to LittleFS
  bot->queueMessage(creds.getChatId(), "Sensor: 25.3°C");

  bot->checkMessages(handleCommands);
  delay(1000);
}
```

Key Differences

Feature	UniversalTelegramBot	KissTelegram
Message loss	Yes (if WiFi fails)	No (LittleFS)
Dependencies	ArduinoJson	None
Memory management	String (leaks)	char[] (no leaks)
Priorities	No	✅ 4 levels
Power management	No	✅ 6 modes
Integrated OTA	No	✅ Dual-bank
Uptime >49 days	❌ Overflow	✅ Managed
Intelligent WiFi	No	✅ Race-safe

Key Features

1. LittleFS Persistence

All messages are saved to flash before sending:

```
cpp

bot->queueMessage(chat_id, "Temperature alert", PRIORITY_CRITICAL);
// ✅ Saved to /queue_messages.json BEFORE attempting to send
```

2. Message Priorities

```
cpp

PRIORITY_LOW    // Logs, debug
PRIORITY_NORMAL // Normal messages (default)
PRIORITY_HIGH   // Alerts
PRIORITY_CRITICAL // Emergencies - activates TURBO mode
```

3. Power Management

6 automatic modes based on activity:

```
cpp

POWER_BOOT    // Starting up (30s)
POWER_LOW     // No messages >5min
POWER_IDLE    // No recent messages
POWER_ACTIVE  // Processing messages
POWER_TURBO   // CRITICAL messages
POWER_MAINTENANCE // During OTA
```

4. Intelligent WiFi Manager

Detects connection quality and waits for stability:

```
cpp
```

QUALITY_DEAD // No connection
QUALITY_POOR // Connected but unstable
QUALITY_FAIR // Acceptable connection
QUALITY_GOOD // Stable connection >30s
QUALITY_EXCELLENT // Perfect connection

Technical Architecture

char[] vs String: Eliminating Memory Leaks

Problem with String (UniversalTelegramBot and other libs):

```
cpp
// ❌ TYPICAL CODE WITH String
void processMessage() {
    String message = "Temperature: ";
    message += String(temp);
    message += "°C";

    String response = bot.sendMessage(chatId, message, "");

    // ❌ PROBLEMS:
    // 1. Heap fragmentation
    // 2. Memory leaks in concatenations
    // 3. Constant reallocs
    // 4. Crashes after hours/days of operation
}

// After 100 iterations:
Serial.printf("Heap: %d bytes\n", ESP.getFreeHeap());
// Heap gradually DECREASES
// Fragmentation increases
// System UNSTABLE after 24-48 hours
```

KissTelegram solution with char[]:

```
cpp
// ✅ KISSTELEGRAM CODE
void processMessage() {
    char message[128];
    snprintf(message, sizeof(message), "Temperature: %.1f°C", temp);

    bot->queueMessage(chatId, message);

    // ✅ ADVANTAGES:
    // 1. Stack allocation (automatic deallocation)
    // 2. Zero fragmentation
    // 3. Zero memory leaks
    // 4. STABLE 24/7/365
}

// After 1000 iterations:
Serial.printf("Heap: %d bytes\n", ESP.getFreeHeap());
// Heap CONSTANT
// Zero fragmentation
// System STABLE indefinitely
```

Real Comparison:

Metric	String (common libs)	char[] (KissTelegram)
Heap after 1h	-15KB	0KB
Heap after 24h	-50KB+	0KB
Fragmentation	High (fatal)	None
Crashes	Frequent	None
Max uptime	2-7 days	Indefinite

KissTelegram Internal Implementation:

cpp

```
// system_setup.h - Pre-sized buffers
#define JSON_BUFFER_SIZE 8192
#define MESSAGE_BUFFER_SIZE 2048

class KissTelegram {
private:
    // ✅ Static buffers in heap (allocated once)
    char* jsonBuffer;
    char* messageBuffer;
    char* commandBuffer;
    char* paramBuffer;

    // Constructor
    KissTelegram(const char* token) {
        // SINGLE allocation at startup
        jsonBuffer = new char[JSON_BUFFER_SIZE];
        messageBuffer = new char[MESSAGE_BUFFER_SIZE];
        commandBuffer = new char[COMMAND_BUFFER_SIZE];
        paramBuffer = new char[PARAM_BUFFER_SIZE];

        // ✅ Never more allocs/deallocs
        // ✅ Zero fragmentation
    }

    void resetBuffers() {
        // ✅ Only memset, no free/malloc
        memset(jsonBuffer, 0, JSON_BUFFER_SIZE);
        memset(messageBuffer, 0, MESSAGE_BUFFER_SIZE);
    }
};
```

Recommendations for your code:

```
cpp

// ❌ AVOID
String buildMessage(float temp) {
    String msg = "Temp: ";
    msg += String(temp);
    return msg; // Memory leak
}

// ✅ USE
void buildMessage(char* buffer, size_t bufSize, float temp) {
    snprintf(buffer, bufSize, "Temp: %.1f°C", temp);
}

// Usage:
char msg[100];
buildMessage(msg, sizeof(msg), 25.3);
bot->queueMessage(chatId, msg);
```

millis() Overflow Management: 24/7 Operation

millis() problem in production:

```
cpp

// ❌ TYPICAL CODE (fails after 49 days)
unsigned long lastCheck = millis();


void loop() {
    unsigned long now = millis();

    // ❌ FAILS when millis() overflows (49.7 days)
    if(now - lastCheck > 60000) {
        checkSensor();
        lastCheck = now;
    }
}

// Failure example:
// Day 49: lastCheck = 4294967295 (near overflow)
// Day 50: now = 100 (overflow occurred)
// now - lastCheck = 100 - 4294967295 = HUGE NEGATIVE
// ❌ Condition NEVER met
```


KissTelegram Solution:

cpp

```
//  KissTelegram.cpp - Safe function
unsigned long KissTelegram::safeTimeDiff(unsigned long later, unsigned long earlier) {
    // Correctly handles 32-bit overflow
    return (later >= earlier)
        ? (later - earlier) // Normal case
        : ((0xFFFFFFFF - earlier) + later + 1); // Overflow case
}


// Internal usage:
void KissTelegram::processQueue() {
    unsigned long now = millis();
    unsigned long timeSinceLastMsg = safeTimeDiff(now, lastMessageTime);

    if(timeSinceLastMsg < minMessageInterval) return;

    //  Works ALWAYS, even after 49+ days
}
```


Practical example for your code:

cpp

```
//  COPY this function to your sketch
unsigned long safeTimeDiff(unsigned long later, unsigned long earlier) {
    return (later >= earlier)
        ? (later - earlier)
        : ((0xFFFFFFFF - earlier) + later + 1);
}

// Use in field sensors:
unsigned long lastSensorRead = 0;

void loop() {
    unsigned long now = millis();


    //  Safe for 24/7/365
    if(safeTimeDiff(now, lastSensorRead) > 3600000) { // 1 hour
        float temp = readSensor();
        bot->queueMessage(chatId, String(temp).c_str());
        lastSensorRead = now;
    }
}
```

Overflow validation:

cpp

```
// Simulated test (no need to wait 49 days)
void testOverflow() {
    unsigned long testCases[][3] = {
        // {earlier, later, expected_diff}
        {100, 200, 100}, // Normal
        {4294967295, 100, 101}, // Overflow
        {4294967000, 1000, 1296}, // Partial overflow
    };

    for(int i = 0; i < 3; i++) {
        unsigned long diff = safeTimeDiff(testCases[i][1], testCases[i][0]);
        Serial.printf("Test %d: %lu (expected %lu) %s\n",
            i, diff, testCases[i][2],
            diff == testCases[i][2] ? " " : " ");
    }
}

// Output:
// Test 0: 100 (expected 100) 
// Test 1: 101 (expected 101) 
// Test 2: 1296 (expected 1296) 
```

Internal KissTelegram usage:

cpp

```

// All timeouts use safeTimeDiff:
bool shouldSave() {
    return (safeTimeDiff(millis(), lastSaveTime) > 300000);
}

bool shouldCleanup() {
    if(safeTimeDiff(millis(), lastCleanupTime) > KISS_CLEANUP_INTERVAL_MS)
        return true;
    return false;
}

void updatePowerState() {
    unsigned long inactiveTime = safeTimeDiff(millis(), lastActivityTime);
    if(inactiveTime > (idleTimeout * 1000)) {
        setPowerMode(POWER_LOW);
    }
}

```

isWifiStable(): Avoiding Race Conditions

Common problem - WiFi Race Conditions:

```

cpp

// ❌ TYPICAL CODE
void loop() {
    // Check 1: WiFi seems OK
    if(WiFi.status() == WL_CONNECTED) {

        // ❌ PROBLEM: WiFi can drop HERE
        // (between check and connect)

        if(client.connect("api.telegram.org", 443)) {
            client.print("GET /bot...");
            // ❌ Connection failed
            // ❌ Message lost
            // ❌ CPU wasted on retries
        }
    }
}

// Symptoms:
// - "Connection failed" in serial
// - Lost messages
// - CPU at 100% retrying
// - Watchdog resets

```

KissTelegram Solution - Intelligent WiFi Manager:

```

cpp

```

```
// 🟢 KissTelegram.cpp
bool KissTelegram::isWifiStable() {
    return enabled && isConnectionReallyStable();
}

bool KissTelegram::isConnectionReallyStable() {
    // 1. Basic check
    if(WiFi.status() != WL_CONNECTED) {
        currentQuality = QUALITY_DEAD;
        return false;
    }

    // 2. Verify initialization
    if(wifiStableTime == 0) return false;

    // 3. Real connectivity test
    if(!testBasicConnectivity()) {
        failedPings++;
        currentQuality = QUALITY_POOR;
        return false;
    }

    // 4. Check stability time (30s by default)
    unsigned long timeStable = safeTimeDiff(millis(), wifiStableTime);
    bool stable = (timeStable > wifiStabilityThreshold);

    if(stable) {
        currentQuality = (failedPings == 0) ? QUALITY_GOOD : QUALITY_FAIR;
    }

    return stable;
}

bool KissTelegram::testBasicConnectivity() {
    WiFiClient testClient;
    if(testClient.connect("www.google.com", 80)) {
        testClient.stop();
        return true;
    }
    return false;
}
```

Usage in your code:

```
cpp

void loop() {
    // 🟢 Intelligent check with multiple validations
    if(bot->isWifiStable()) {
        // WiFi REALLY stable (30s+ connected + ping OK)
        bot->processQueue();
        bot->checkMessages(handleCommands);
    } else {
        // WiFi unstable or recently connected
        // 🟢 DON'T waste CPU on requests that will fail
        Serial.println(" 🕒 Waiting for stable WiFi...");
    }

    delay(1000);
}
```

System advantages:

```
cpp
```

```
// 1. Avoids unnecessary work
if(!bot->isWifiStable()) {
    // ✅ DON'T execute:
    // - HTTP requests
    // - Telegram API calls
    // - Queue processing
    // → CPU and battery savings
    return;
}

// 2. Gradual connection quality
switch(bot->getConnectionQuality()) {
    case QUALITY_DEAD:
        // WiFi down
        break;

    case QUALITY_POOR:
        // Connected but pings fail
        // ✅ Wait before sending
        break;

    case QUALITY_FAIR:
        // Acceptable connection
        // ✅ OK for NORMAL messages
        break;

    case QUALITY_GOOD:
        // Stable >30s, no failures
        // ✅ OK for EVERYTHING
        break;
}

// 3. Automatic reconnection management
void checkWifiStability() {
    static bool wasConnected = false;
    bool isConnected = (WiFi.status() == WL_CONNECTED);

    if(wasConnected && !isConnected) {
        // WiFi DOWN
        bot->disable(); // ✅ Stop processing
        Serial.println(" 📶 WiFi disconnected");
    }

    if(!wasConnected && isConnected) {
        // WiFi RECOVERED
        Serial.println(" 📶 WiFi recovered - stabilizing...");
        delay(5000); // Wait for stabilization
        bot->setWifiStable();
        bot->enable(); // ✅ Reactivate processing
    }

    wasConnected = isConnected;
}
```

Behavior comparison:

Scenario	Typical code	KissTelegram
WiFi drops during send	❌ Crash/hang	✅ Detects and stops
WiFi reconnection	❌ Immediate send (fails)	✅ Wait 30s stable
Ping fails	❌ Doesn't detect	✅ Marks QUALITY_POOR
Wasted CPU	❌ Constant retries	✅ Only if stable

Field Power Management

Why Power Management in IoT?

Typical scenario - Remote sensor:

Location: Mountain weather station
Power: Solar panel + battery
Connection: Intermittent 4G/LTE
Requirement: 365 days/year uptime

Problem without Power Management:

cpp


```
// ❌ CODE WITHOUT POWER MANAGEMENT

void loop() {
  // CPU ALWAYS at 100%
  checkWiFi();      // 30% CPU
  processQueue();    // 40% CPU
  checkMessages();   // 30% CPU

  // Result:
  // - Battery lasts 2-3 days with panel
  // - No sunlight → shutdown in 12 hours
  // - Constant WiFi scanning → interference

  delay(100); // CPU idle only 10% of time
}

// Typical consumption: 150-200mA average
// 5000mAh battery → ~25 hours without sun
```

KissTelegram Solution - Power Management:

```
cpp

// ✅ CODE WITH POWER MANAGEMENT

void loop() {
  // CPU adapts to workload
  bot->updatePowerState(); // Evaluate needed mode

  if(bot->shouldProcessQueue()) {
    bot->processQueue();
  }

  if(bot->shouldCheckMessages()) {
    bot->checkMessages(handleCommands);
  }

  // Adaptive delay based on mode
  int delayTime = bot->getRecommendedDelay();
  delay(delayTime); // 500ms - 10000ms depending on mode

  // Result:
  // - Battery lasts 7-15 days with panel
  // - No sunlight → 3-5 days autonomy
  // - WiFi scanning only when necessary
}

// Typical consumption: 30-80mA average (60% reduction)
// 5000mAh battery → ~65 hours without sun
```

Power Modes Explained

POWER_BOOT (first 30 seconds):

```
cpp

// Characteristics:
// - WiFi: Initializing
// - Queue: Not processing
// - Check messages: Every 30s
// - Recommended delay: 5000ms

// When?
// - Booting after reset
// - Connecting WiFi
// - Waiting for stabilization

// Consumption: ~120mA
```

POWER_LOW (no activity >5 min):

```
cpp
```

```
// Characteristics:
// - WiFi: Maintains connection
// - Queue: Not processing
// - Check messages: Every 60s
// - Recommended delay: 10000ms

// When?
// - No pending messages
// - No recent commands
// - Night time (optional)

// Consumption: ~30mA
// → Ideal for night savings

// Configuration example:
void setup() {
    bot->setPowerConfig(
        300, // 5 min idle → POWER_LOW
        10,  // 10s transition
        30   // 30s boot stable
    );
}
```

POWER_IDLE (no recent activity):

```
cpp

// Characteristics:
// - WiFi: Active
// - Queue: Process every 3s
// - Check messages: Every 15s
// - Recommended delay: 3000ms

// When?
// - Empty queue
// - No commands in 1-5 min
// - Waiting for events

// Consumption: ~60mA
// → Balance savings/response
```

POWER_ACTIVE (processing messages):

```
cpp

// Characteristics:
// - WiFi: Active
// - Queue: Process every 1s
// - Check messages: Every 10s
// - Recommended delay: 1000ms

// When?
// - Messages in queue
// - Sending periodically
// - Recent commands (<1 min)

// Consumption: ~100mA
// → Normal operation mode
```

POWER_TURBO (emergencies):

```
cpp
```

```
// Characteristics:
// - WiFi: Maximum priority
// - Queue: Process every 500ms
// - Check messages: Every 5s
// - Recommended delay: 500ms

// When?
// - PRIORITY_CRITICAL messages
// - System alerts
// - Failure recovery

// Consumption: ~150mA
// → Only for emergencies

// Automatic trigger example:
bot->queueMessage(chatId, "🔥 ALARM", PRIORITY_CRITICAL);
// ✅ Activates POWER_TURBO automatically
// ✅ Sends IMMEDIATELY
// ✅ Returns to ACTIVE after sending
```

POWER_MAINTENANCE (during OTA):

```
cpp

// Characteristics:
// - WiFi: Maximum stability
// - Queue: Only OTA messages
// - Check messages: Only OTA commands
// - Recommended delay: Variable

// When?
// - Active OTA process
// - Downloading firmware
// - Validating update

// Consumption: ~120mA
// → Necessary for stable OTA

// Automatic activation:
bot->setMaintenanceMode(true, "OTA in progress");
// ✅ Pauses normal messages
// ✅ Allows only OTA
// ✅ Ensures stability
```

Configuration for Field Deployment

Autonomous weather station:

```
cpp
```

```

void setup() {
    // Aggressive power management
    bot->setPowerSaving(true);
    bot->setPowerConfig(
        600, // 10 min idle → POWER_LOW
        15,  // 15s transitions
        45   // 45s boot stable
    );

    // Power save operation mode
    bot->setOperationMode(MODE_POWERSAVE);
    // - MinMessageInterval: 2000ms
    // - Reduced WiFi checks
    // - Maximized CPU idle
}

void loop() {
    // Update every hour (except emergencies)
    static unsigned long lastReport = 0;
    if(safeTimeDiff(millis(), lastReport) >= 3600000) {
        float temp = readTemp();
        float hum = readHumidity();

        char msg[150];
        snprintf(msg, sizeof(msg),
            "🌡️ Hourly report\n"
            "🌡️ %s, 1f°\n"
            "💧 %s, 1f%%",
            temp, hum);

        bot->queueMessage(chatId, msg, PRIORITY_NORMAL);
        lastReport = millis();
    }

    // Critical alerts (bypass power management)
    if(temp >= 50.0) {
        bot->queueMessage(chatId,
            "🔥 Critical temperature",
            PRIORITY_CRITICAL);
        // ✅ Activates POWER_TURBO
        // ✅ Immediate sending
    }

    // Intelligent management
    bot->updatePowerState();

    if(bot->shouldProcessQueue()) {
        bot->processQueue();
    }

    if(bot->shouldCheckMessages()) {
        bot->checkMessages(handleCommands);
    }

    delay(bot->getRecommendedDelay());
    // POWER_LOW: 10000ms → CPU idle 90%
    // POWER_ACTIVE: 1000ms → CPU idle 30%
    // ✅ Battery lasts 5x longer
}

```

Agricultural sensor with solar:

cpp

```

void setup() {
    // Balance savings/response
    bot->setOperationMode(MODE_BALANCED);

    // Callbacks for monitoring
    bot->onPowerModeChange([](PowerMode old, PowerMode nuevo) {
        Serial.printf("Power: %d → %d\n", old, nuevo);

        // Log transitions
        char log[80];
        snprintf(log, sizeof(log),
            " ⚡ Power mode: %d → %d", old, nuevo);
        bot->queueMessage(chatId, log, PRIORITY_LOW);
    });
}

void loop() {
    // Measurements every 15 min
    static unsigned long lastMeasure = 0;
    if(safeTimeDiff(millis(), lastMeasure) > 900000) {

        float soilHumidity = readSoil();

        if(soilHumidity < 20.0) {
            // 🚰 Needs irrigation
            bot->queueMessage(chatId,
                " 🚰 Dry soil - irrigation needed",
                PRIORITY_HIGH);
            // ✅ Rises to POWER_ACTIVE for fast sending
        } else {
            // 📺 Normal report
            char msg[100];
            snprintf(msg, sizeof(msg),
                " 📺 Soil humidity: %.1f%%", soilHumidity);
            bot->queueMessage(chatId, msg, PRIORITY_NORMAL);
            // ✅ Sends when WiFi and battery allow
        }

        lastMeasure = millis();
    }

    // Automatic management
    bot->updatePowerState();

    // Only process if it makes sense
    if(bot->shouldProcessQueue()) {
        bot->processQueue();
    }

    if(bot->shouldCheckMessages()) {
        bot->checkMessages(handleCommands);
    }

    // Additional night savings (optional)
    struct tm timeinfo;
    if(getLocalTime(&timeinfo)) {
        if(timeinfo.tm_hour >= 22 || timeinfo.tm_hour <= 6) {
            // Night: force POWER_LOW
            bot->setPowerMode(POWER_LOW);
        }
    }

    delay(bot->getRecommendedDelay());
}

```

Industrial 24/7 monitoring:

cpp

```
void setup() {
    // Maximum response
    bot->setOperationMode(MODE_PERFORMANCE);

    // Minimum power management (mains powered)
    bot->setPowerSaving(false);
    // ✅ Always in POWER_ACTIVE or POWER_TURBO
}

void loop() {
    // Continuous monitoring
    float pressure = readPressure();
    float flow = readFlow();

    if(pressure > MAX_PRESSURE) {
        // 🚨 Critical alert
        bot->queueMessage(chatId,
            "🚨 CRITICAL PRESSURE - Automatic shutdown",
            PRIORITY_CRITICAL);

        shutdownSystem(); // Safety
    }

    // Report every 5 min (even without power management)
    static unsigned long lastReport = 0;
    if(safeTimeDiff(millis(), lastReport) > 300000) {
        char msg[200];
        snprintf(msg, sizeof(msg),
            "🏭 PLANT STATUS\n"
            "⚙️ Pressure: %.1f bar\n"
            "💧 Flow: %.1f L/min\n"
            "🕒 Uptime: %lu h",
            pressure, flow,
            millis() / 3600000);

        bot->queueMessage(chatId, msg, PRIORITY_HIGH);
        lastReport = millis();
    }

    // No power management → always processes
    bot->processQueue();
    bot->checkMessages(handleCommands);

    delay(500); // Fast response
}
```

Power Management Statistics

cpp

```
void printPowerStats() {
  Serial.println("\n ⚡ POWER STATISTICS:");

  const char* names[] = {
    "BOOT", "LOW", "IDLE", "ACTIVE", "TURBO", "MAINT"
  };

  for(int i = 0; i < 6; i++) {
    unsigned long timeInMode = bot->getTimeInMode((PowerMode)i);
    Serial.printf(" - %s: %lu min\n",
      names[i],
      timeInMode / 60000);
  }

  float efficiency = bot->getPowerEfficiency();
  Serial.printf("\n 📊 Efficiency: %.2f msg/s\n", efficiency);

  // Example output after 24h:
  // ⚡ POWER STATISTICS:
  // - BOOT: 1 min
  // - LOW: 720 min (12h night)
  // - IDLE: 500 min
  // - ACTIVE: 218 min
  // - TURBO: 1 min
  // - MAINT: 0 min
  //
  // 📊 Efficiency: 0.08 msg/s
  // ✅ 50% time in LOW/IDLE → 60% energy savings
}

// Call every 6 hours:
static unsigned long lastStats = 0;
if(safeTimeDiff(millis(), lastStats) > 21600000) {
  printPowerStats();
  bot->resetPowerStatistics(); // Reset counters
  lastStats = millis();
}
```

Installation and Setup

Step 1: Hardware Requirements

- ESP32 board with PSRAM (minimum 8MB)
- Flash minimum 4MB (recommended 16MB)

Step 2: Partition Table

Edit your ESP32 `(boards.txt)` to include:

```
esp32.menu.PartitionScheme.kiss=KissTelegram (13MB LittleFS)
esp32.menu.PartitionScheme.kiss.build.partitions=partitions
esp32.menu.PartitionScheme.kiss.upload.maximum_size=1572864
```

Step 3: Configure Credentials

Edit `(system_setup.h)`:

```
cpp

#define KISS_FALLBACK_BOT_TOKEN "123456:ABC-DEF..."
#define KISS_FALLBACK_CHAT_ID "615715406"
#define KISS_FALLBACK_WIFI_SSID "MyWiFi"
#define KISS_FALLBACK_WIFI_PASSWORD "mypassword"
#define KISS_FALLBACK_OTA_PIN "1234"
#define KISS_FALLBACK_OTA_PUK "12345678"
```

Step 4: Basic Code

```
cpp
```

```

#include "system_setup.h"
#include "KissCredentials.h"
#include "KissTelegram.h"

KissCredentials& creds = KissCredentials::getInstance();
KissTelegram* bot;

void setup() {
  Serial.begin(115200);

  // 1. Initialize credentials
  creds.begin();

  // 2. Connect WiFi
  WiFi.begin(creds.getWifiSSID(), creds.getWifiPassword());
  while (WiFi.status() != WL_CONNECTED) delay(500);

  // 3. Create bot and restore messages
  bot = new KissTelegram(creds.getBotToken());
  bot->restoreFromStorage(); // ✅ Recovers pending messages

  // 4. Activate
  bot->setWifiStable();
  bot->enable();

  Serial.printf("✅ Restored messages: %d\n", bot->getMessagesInFS());
}

void loop() {
  bot->processQueue(); // Send pending messages
  bot->checkMessages(handleCommands);
  delay(1000);
}

void handleCommands(const char* chat_id, const char* text,
                    const char* command, const char* param) {
  if(strcmp(command, "/start") == 0) {
    bot->sendMessage(chat_id, "🤖 Bot active");
  }
}

```

Message Persistence - Zero Loss

Scenario 1: WiFi Outage

Common problem (UniversalTelegramBot):

```

cpp

// ❌ If WiFi fails here, message is lost
bot.sendMessage(CHAT_ID, "Sensor alert", "");

```

KissTelegram Solution:

```

cpp

// ✅ Message saved to LittleFS immediately
bot->queueMessage(chat_id, "Sensor alert", PRIORITY_HIGH);

// WiFi drops...
// ✅ Message still in /queue_messages.json

// WiFi returns...
bot->processQueue(); // ✅ Sends automatically

```

Scenario 2: Unexpected Reboot

Simulation:

```

cpp

```



```
void loop() {
  // User sends 10 messages
  for(int i = 0; i < 10; i++) {
    char msg[50];
    sprintf(msg, sizeof(msg), "Message %d", i);
    bot->queueMessage(chat_id, msg);
  }

  // ❌ ESP32 reboots HERE (power outage)
  ESP.restart();
}

// After reboot:
void setup() {
  bot = new KissTelegram(BOT_TOKEN);
  bot->restoreFromStorage();
  // ✅ All 10 messages still there
  Serial.printf("Pending messages: %d\n", bot->getMessagesInFS());
}
```

Scenario 3: OTA Update

```
cpp

// User has 50 pending messages in LittleFS
Serial.printf("Pending before OTA: %d\n", bot->getMessagesInFS());

// Start OTA with /ota
// During OTA → maintenance mode active
// ✅ No messages lost

// After update:
bot->restoreFromStorage();
Serial.printf("Pending after OTA: %d\n", bot->getMessagesInFS());
// ✅ Still 50 messages
```

Verify Persistence

Manual test:

```
cpp

void testPersistence() {
  // 1. Queue 20 messages
  for(int i = 0; i < 20; i++) {
    char msg[50];
    sprintf(msg, sizeof(msg), "Persistence test %d", i);
    bot->queueMessage(chat_id, msg);
  }

  // 2. Check status
  Serial.printf("In FS: %d\n", bot->getMessagesInFS());

  // 3. Reboot manually
  delay(2000);
  ESP.restart();

  // 4. In setup() after reboot:
  bot->restoreFromStorage();
  Serial.printf("Recovered: %d\n", bot->getMessagesInFS());
  // ✅ Should show 20
}
```

Useful commands:

```
/debugfs → View LittleFS contents
/storage → Storage statistics
/cleanup → Clean sent messages
/borrar confirmar → Delete all
```

Priority Management

Priority Levels

```
cpp
```

```
// 🟢 LOW - Logs, debug (doesn't activate power management)
bot->queueMessage(chat_id, "Debug: Variable X = 42", PRIORITY_LOW);

// 🔵 NORMAL - Normal messages (default)
bot->queueMessage(chat_id, "Sensor: 25.3°C");

// 🟡 HIGH - Alerts
bot->queueMessage(chat_id, " ⚠️ High temperature", PRIORITY_HIGH);

// 🔴 CRITICAL - Emergencies (activates TURBO mode)
bot->queueMessage(chat_id, " 🚨 CRITICAL FAILURE", PRIORITY_CRITICAL);
```

Example: Sensor with Alerts

```
cpp

void monitorTemperature() {
    float temp = readSensor();

    if(temp > 80) {
        // 🔴 Emergency → jumps entire queue
        char msg[100];
        snprintf(msg, sizeof(msg), " 🚨 CRITICAL TEMPERATURE: %.1f°C", temp);
        bot->queueMessage(chat_id, msg, PRIORITY_CRITICAL);

    } else if(temp > 60) {
        // 🟡 Alert → priority over normal messages
        char msg[100];
        snprintf(msg, sizeof(msg), " ⚠️ High temperature: %.1f°C", temp);
        bot->queueMessage(chat_id, msg, PRIORITY_HIGH);

    } else {
        // 🔵 Normal info
        char msg[100];
        snprintf(msg, sizeof(msg), " 📺 Temperature: %.1f°C", temp);
        bot->queueMessage(chat_id, msg);
    }
}
```

Priority Test

Command `/prioridades` executes:

```
cpp

void testPriorities() {
    // Queue in reverse priority order
    bot->queueMessage(chat_id, " 🟢 LOW message", PRIORITY_LOW);
    bot->queueMessage(chat_id, " 🔵 NORMAL message", PRIORITY_NORMAL);
    bot->queueMessage(chat_id, " 🟡 HIGH message", PRIORITY_HIGH);
    bot->queueMessage(chat_id, " 🔴 CRITICAL message", PRIORITY_CRITICAL);

    // ✅ Sent in order: CRITICAL → HIGH → NORMAL → LOW
}
```

System Commands

Basic Commands

```
/help or /start
└─ Shows complete help

/estado
└─ System status:
    • Uptime
    • Messages sent/pending
    • Free memory
    • Power mode
    • Auto-messages ON/OFF

/config
└─ Configuration to serial monitor

/stats
└─ Cumulative statistics (lifetime):
    • Total queued
    • Total sent
```

- Success rate
- Memory

Diagnostic Commands

/debug
└─ Detailed info to serial

/debugfs
└─ LittleFS analysis:

- Pending messages (s:0)
- Sent messages (s:1)
- Shows JSON in serial

/memoria
└─ Memory info:

- Free heap
- Minimum heap
- LittleFS used

/storage
└─ Storage status

Management Commands

/cleanup
└─ Clean sent messages from FS

/llenar [N]
└─ Test: queue N messages (default 15)

/forceprocess
└─ Force queue sending

/borrar confirmar
└─ 🚨 Delete ALL pending messages

/prioridades
└─ Priority system test

/testlittlefs
└─ Persistence test (generates 10 msg)

System Commands

/parar
└─ Disable automatic messages

/activar
└─ Reactivate automatic messages

/changePIN <old> <new>
└─ Change OTA PIN

/changePUK <old> <new>
└─ Change OTA PUK

/credentials
└─ View credentials in serial

/resetcredentials CONFIRMAR
└─ 🚨 Reset to defaults

OTA Updates

Complete Process

1. Start OTA

User: /ota
Bot: 📡 OTA UPDATE

During update:

- Messages remain stored
- You won't lose messages
- You won't receive messages

Type /otapin [PIN] to start
⌚ Timeout: 60 seconds

2. Authenticate with PIN

User: /otapin 1234
Bot: ✅ CORRECT PIN

Starting process...

- Messages paused
- Checking space
- Creating backup

⌚ Process timeout: 7 minutes

3. Send Firmware

- Send `.bin` file via Telegram
- Bot downloads to PSRAM and verifies checksum

Bot: 📡 DOWNLOADING FIRMWARE

Receiving: firmware_v2.bin
Size: 1.2 MB

Do not power off device.

4. Confirm Flash

Bot: ✅ FIRMWARE VERIFIED

File: firmware_v2.bin
Size: 1.2 MB
CRC32: 0x12345678

⚠️ CONFIRM FLASH

/otaconfirm - Flash now
/otacancel - Cancel process

⌚ Timeout: 3 minutes

User: /otaconfirm

Bot: 🔥 FLASHING FIRMWARE

Writing to flash memory...
⚠️ DO NOT POWER OFF DEVICE

5. Validate after Reboot

[ESP32 reboots automatically]

Bot: 📡 FIRMWARE VALIDATION

Device has booted.

🕒 You have 60 seconds to validate

/otaok - Confirm it works

/otacancel - Force rollback

If no response, automatic rollback will execute.

User: /otaok

Bot: ✅ FIRMWARE VALIDATED

Update completed successfully.

Backup deleted.

System running on new firmware.

(You can ignore the previous validation message)

Automatic Rollback

If you DON'T send (/otaok) within 60 seconds:

[60 seconds pass without /otaok]

Bot: 🕒 VALIDATION TIMEOUT

No confirmation received in 60 seconds.

Starting automatic rollback...

Device will reboot with previous firmware.

[ESP32 reboots with original firmware]

Bot: ✅ Rollback complete

System running on previous firmware

Manual Rollback

If you detect problems after updating:

User: /otacancel

Bot: 📡 STARTING ROLLBACK

Restoring previous firmware...

Device will reboot.

Do not power off device.

[ESP32 reboots with original firmware]

OTA Security

PIN lockout:

User: /otapin 0000

Bot: ❌ Incorrect PIN

Remaining attempts: 2

User: /otapin 1111

Bot: ❌ Incorrect PIN

Remaining attempts: 1

User: /otapin 2222

Bot: 🚫 PIN LOCKED

Maximum 3 attempts exceeded.

To unlock type /otapuk [PUK]

PUK unlock:

User: /otapuk 12345678
Bot: 📶 OTA UNLOCKED

PIN has been unlocked.
Attempts have been reset.

To start update type /ota

Practical Examples

Example 1: Temperature Sensor (Field Optimized)

```
cpp

#include "KissTelegram.h"
#include "KissCredentials.h"

KissCredentials& creds = KissCredentials::getInstance();
KissTelegram* bot;

void setup() {
    creds.begin();
    WiFi.begin(creds.getWifiSSID(), creds.getWifiPassword());
    while (WiFi.status() != WL_CONNECTED) delay(500);

    bot = new KissTelegram(creds.getBotToken());
    bot->restoreFromStorage();
    bot->setWifiStable();
    bot->enable();

    // ✅ Power management for field
    bot->setPowerSaving(true);
    bot->setOperationMode(MODE_POWERSAVE);
}

void loop() {
    // ✅ Only process if WiFi really stable
    if(bot->isWifiStable()) {
        bot->processQueue();
    }

    // Read sensor every 5 minutes
    static unsigned long lastRead = 0;
    unsigned long now = millis();

    // ✅ Use safeTimeDiff for uptime >49 days
    if(safeTimeDiff(now, lastRead) > 300000) {
        float temp = readTemperature();

        // ✅ char[] instead of String
        char msg[100];
        sprintf(msg, sizeof(msg), "🌡️ Temperature: %.1f°C", temp);

        // ✅ If WiFi fails, message saved to LittleFS
        bot->queueMessage(creds.getChatId(), msg);

        lastRead = now;
    }

    // ✅ Adaptive delay based on power mode
    delay(bot->getRecommendedDelay());
}

// ✅ Safe function for millis overflow
unsigned long safeTimeDiff(unsigned long later, unsigned long earlier) {
    return (later >= earlier)
        ? (later - earlier)
        : ((0xFFFFFFFF - earlier) + later + 1);
}
```

Example 2: Alert System (No Memory Leaks)

```
cpp
```

```

void monitorSystem() {
    // Read sensors
    float temp = readTemperature();
    float humidity = readHumidity();
    int battery = readBattery();

    // ✅ Pre-sized char[] buffers
    char msg[200];

    // Critical temperature
    if(temp > 80) {
        snprintf(msg, sizeof(msg),
            "🔥 CRITICAL ALERT\n\n"
            "Temperature: %.1f°C\n"
            "Battery: %d%%\n"
            "Immediate action required",
            temp, battery);
        bot->queueMessage(creds.getChatId(), msg, PRIORITY_CRITICAL);
        // ✅ Activates POWER_TURBO automatically
    }

    // High temperature
    else if(temp > 60) {
        snprintf(msg, sizeof(msg),
            "⚠️ High temperature: %.1f°C", temp);
        bot->queueMessage(creds.getChatId(), msg, PRIORITY_HIGH);
    }

    // Low battery
    if(battery < 20) {
        snprintf(msg, sizeof(msg),
            "🔋 Low battery: %d%%", battery);
        bot->queueMessage(creds.getChatId(), msg, PRIORITY_HIGH);
    }

    // Normal report every hour
    static unsigned long lastReport = 0;
    unsigned long now = millis();

    // ✅ safeTimeDiff for uptime >49 days
    if(safeTimeDiff(now, lastReport) > 3600000) {
        snprintf(msg, sizeof(msg),
            "📊 HOURLY REPORT\n\n"
            "🌡️ Temp: %.1f°C\n"
            "💧 Humidity: %.1f%%\n"
            "🔋 Battery: %d%%\n"
            "🕒 Uptime: %lu min",
            temp, humidity, battery,
            now / 60000);
        bot->queueMessage(creds.getChatId(), msg, PRIORITY_NORMAL);
        lastReport = now;
    }
}

// ✅ Zero memory leaks
// ✅ Constant heap after 1000+ iterations

```

Example 3: Remote Control (Intelligent WiFi)

cpp

```

void handleCommands(const char* chat_id, const char* text,
                    const char* command, const char* param) {

    // ✅ char[] buffers for responses
    char response[300];

    // Turn on LED
    if(strcmp(command, "/led") == 0) {
        if(strcmp(param, "on") == 0) {
            digitalWrite(LED_PIN, HIGH);
            bot->sendMessage(chat_id, "💡 LED on");
        }
        else if(strcmp(param, "off") == 0) {
            digitalWrite(LED_PIN, LOW);
            bot->sendMessage(chat_id, "💡 LED off");
        }
    }

    // Read sensor on demand
    else if(strcmp(command, "/temp") == 0) {
        float temp = readTemperature();
        snprintf(response, sizeof(response),
                 "🌡️ Current temperature: %.1f°C", temp);
        bot->sendMessage(chat_id, response);
    }

    // Restart device
    else if(strcmp(command, "/restart") == 0) {
        bot->sendMessage(chat_id, "🔄 Restarting...");
        delay(2000);
        ESP.restart();
    }

    // System status
    else if(strcmp(command, "/status") == 0) {
        unsigned long uptime = millis();

        snprintf(response, sizeof(response),
                 "🖨️ SYSTEM STATUS\n\n"
                 "🕒 Uptime: %lu min\n"
                 "💾 Free heap: %d bytes\n"
                 "📬 Pending messages: %d\n"
                 "🔌 Power Mode: %d\n"
                 "📶 WiFi: %s",
                 uptime / 60000,
                 ESP.getFreeHeap(),
                 bot->getMessagesInFS(),
                 bot->getCurrentPowerMode(),
                 bot->isWifiStable() ? "Connected" : "Disconnected");
        bot->sendMessage(chat_id, response);
    }
}

void loop() {
    // ✅ Only process if WiFi REALLY stable
    // Avoids race conditions and unnecessary work
    if(bot->isWifiStable()) {
        bot->processQueue();
        bot->checkMessages(handleCommands);
    } else {
        Serial.println("⌚ Waiting for stable WiFi...");
    }

    delay(1000);
}

```

Example 4: Logger with Persistence (24/7 Production)

cpp


```

// Logging system that never loses messages
// 🟢 Uptime >49 days without problems

class Logger {
private:
    KissTelegram* bot;
    char chatId[20];

public:
    void init(KissTelegram* b, const char* chat) {
        bot = b;
        strncpy(chatId, chat, sizeof(chatId) - 1);
        chatId[sizeof(chatId) - 1] = '\0';
    }

    void debug(const char* msg) {
        bot->queueMessage(chatId, msg, PRIORITY_LOW);
    }

    void info(const char* msg) {
        bot->queueMessage(chatId, msg, PRIORITY_NORMAL);
    }

    void warning(const char* msg) {
        char fullMsg[250];
        snprintf(fullMsg, sizeof(fullMsg), "⚠️ %s", msg);
        bot->queueMessage(chatId, fullMsg, PRIORITY_HIGH);
    }

    void error(const char* msg) {
        char fullMsg[250];
        snprintf(fullMsg, sizeof(fullMsg), "🔥 %s", msg);
        bot->queueMessage(chatId, fullMsg, PRIORITY_CRITICAL);
    }
};

// Usage:
Logger logger;
unsigned long bootTime;

void setup() {
    logger.init(bot, creds.getChatId());
    bootTime = millis();

    logger.info("System started");
    logger.debug("Configuring sensors...");

    if(!initializeSensors()) {
        logger.error("Failed to initialize sensors");
    }
}




void loop() {
    if(detectError()) {
        logger.warning("Error detected in sensor X");
    }

    // 🟢 All logs saved to LittleFS
    // 🟢 Sent automatically when WiFi is stable
    // 🟢 Survive reboots

    // Statistics every 24h
    static unsigned long lastStats = 0;
    unsigned long now = millis();

    // 🟢 safeTimeDiff for uptime >49 days
    if(safeTimeDiff(now, lastStats) > 86400000) {
        char stats[200];
        snprintf(stats, sizeof(stats),
            "📅 Uptime: %lu days\n"
            "🗑️ Heap: %d bytes\n"
            "📄 Msgs in FS: %d",
            safeTimeDiff(now, bootTime) / 86400000,
            ESP.getFreeHeap(),
            bot->getMessagesInFS());
        logger.info(stats);
        lastStats = now;
    }
}

```

```
}  
}  
  
//  Zero memory leaks  
//  Works >49 days without overflow  
//  Intelligent WiFi avoids race conditions
```

Troubleshooting

Problem: Messages not sending



Diagnosis:

```
/estado    → View pending messages  
/debugfs   → View LittleFS contents  
/debug      → Complete info to serial
```

Possible causes:

1. WiFi not stable → wait for `bot->setWifiStable()`
2. Maintenance mode active → verify with `/estado`
3. Low power mode → messages send slower

Solution:

```
cpp  
  
// Force processing  
bot->processQueue();  
  
// Verify WiFi status  
if(bot->isWifiStable()) {  
    Serial.println("  WiFi OK");  
} else {  
    Serial.println("  WiFi unstable");  
}
```

Problem: "Queue full"

Diagnosis:

```
/storage    → View LittleFS usage  
/debugfs    → Count pending messages
```

Solution:

```
/cleanup     → Clean sent messages  
/borrar confirmar → Delete all (emergency)
```

Problem: Low memory

Diagnosis:

```
/memoria     → View available heap
```

Solution:

```
cpp  
  
// Clean sent messages more frequently  
bot->cleanupStorage();  
  
// Reduce auto-save interval  
bot->setAutoSaveInterval(60000); // 1 minute
```

Problem: System unstable after hours

Diagnosis:

```
cpp  
  
// Check heap in loop  
Serial.printf("Heap: %d\n", ESP.getFreeHeap());
```

Probable cause: Memory leaks from String usage

Solution:

```
cpp
```

```
// ❌ DON'T USE
String msg = "Temp: " + String(temp);

// ✅ USE
char msg[100];
sprintf(msg, sizeof(msg), "Temp: %.1f", temp);
```

Problem: Crashes after 49 days

Cause: Unmanaged millis() overflow

Solution:

```
cpp

// ✅ Use safeTimeDiff throughout code
unsigned long safeTimeDiff(unsigned long later, unsigned long earlier) {
    return (later >= earlier)
        ? (later - earlier)
        : ((0xFFFFFFFF - earlier) + later + 1);
}
```

Problem: OTA won't start

Verify:

1. Correct PIN
2. Space in LittleFS ((/debugfs))
3. PSRAM available

Important logs:

```
📦 OTA request received...
✅ Sufficient space
📁 Starting backup...
```

Problem: Rollback doesn't work

Check in serial:

```
🔄 Validating new firmware...
🕒 You have 60 seconds to validate
```

If doesn't appear → check (isFirstBootAfterOTA()) in code.

Advanced Configuration

Power Management

```
cpp

// Customize timeouts
bot->setPowerConfig(
    300, // idleTimeout: 5 min no messages → POWER_LOW
    10,  // decayTime: 10s for transitions
    30   // bootStableTime: 30s after boot
);

// Disable power saving
bot->setPowerSaving(false);
```

Callbacks

```
cpp

// Power mode change notification
bot->onPowerModeChange([](PowerMode oldMode, PowerMode newMode) {
    Serial.printf("Power: %d → %d\n", oldMode, newMode);
});

// System events
bot->onSystemEvent([](const char* event, const char* data) {
    Serial.printf("Event: %s - %s\n", event, data);
});
```

Storage Configuration

```
cpp
```

```
// Auto-save interval
bot->setAutoSaveInterval(300000); // 5 minutes

// Maximum messages in storage
bot->setMaxQueueStorage(500);

// Compress storage (experimental)
bot->setStorageCompression(true);
```

Conclusion

KissTelegram is designed for **critical IoT deployments** where message loss is not an option. Its LittleFS-based architecture ensures every message reaches its destination, regardless of:

- WiFi quality
- Power outages
- System reboots
- OTA updates

Unique Features:

- ☒ Zero external dependencies
- ☒ Zero memory leaks (pure char[])
- ☒ Zero message loss
- ☒ Safe dual-bank OTA
- ☒ Intelligent power management
- ☒ Uptime >49 days (millis() management)
- ☒ WiFi race-safe

Critical Technical Advantages:

Aspect	Implementation	Benefit
Memory	char[] vs String	Zero leaks, indefinite uptime
Time	safeTimeDiff()	Works >49 days without crash
WiFi	isWifiStable()	Avoids race conditions and CPU waste
Energy	6 power modes	60% battery savings in field
Persistence	LittleFS primary	Zero lost messages

Questions or problems?

- Email: victek@gmail.com
- GitHub Issues: [KissTelegram/issues](https://github.com/KissTelegram/telegram/issues)

Manual Version: 5.0.0

Date: November 2024

Author: Vicente Soriano