





# Manual de Usuario KissTelegram v5.0

## Índice

1. [Introducción](#)
  2. [Migración desde UniversalTelegramBot](#)
  3. [Características Principales](#)
  4. [Arquitectura Técnica](#)
  5. [Instalación y Configuración](#)
  6. [Persistencia de Mensajes - Cero Pérdidas](#)
  7. [Gestión de Prioridades](#)
  8. [Power Management en Campo](#)
  9. [Comandos del Sistema](#)
  10. [Actualización OTA](#)
  11. [Ejemplos Prácticos](#)
  12. [Troubleshooting](#)
- 

## Introducción

**KissTelegram** es un framework ESP32 para bots de Telegram diseñado con un objetivo: **cero pérdida de mensajes**. A diferencia de otras bibliotecas, KissTelegram usa LittleFS como almacenamiento primario, no como backup. Esto garantiza que:

-  Cortes de WiFi → mensajes guardados
-  Reinicios del ESP32 → mensajes guardados
-  Cortes de corriente → mensajes guardados
-  Actualizaciones OTA → mensajes guardados

### ¿Por qué cambiar de UniversalTelegramBot?

- **Persistencia real:** Los mensajes sobreviven a cualquier fallo
  - **Prioridades:** Mensajes críticos saltan la cola
  - **Power Management:** 6 modos de ahorro energético
  - **OTA seguro:** Actualización dual-bank con rollback automático
  - **Sin dependencias:** No necesita ArduinoJson ni bibliotecas externas
  - **Sin memory leaks:** Todo en char[], sin Strings
  - **Producción 24/7:** Gestión correcta de millis() overflow (49+ días)
  - **WiFi inteligente:** Evita race conditions y trabajo innecesario
- 

## Migración desde UniversalTelegramBot

### Comparativa de Código

#### UniversalTelegramBot (antiguo):

```
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() {
  // ❌ Si WiFi falla aquí, el mensaje se pierde
  bot.sendMessage(CHAT_ID, "Sensor: 25.3°C", "");
  delay(60000);
}
```

#### KissTelegram (nuevo):

```
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(); // ✅ Recupera mensajes pendientes
  bot->setWifiStable();
  bot->enable();
}

void loop() {
  bot->processQueue(); // ✅ Envía mensajes pendientes

  // ✅ Si WiFi falla, el mensaje se guarda en LittleFS
  bot->queueMessage(creds.getChatId(), "Sensor: 25.3°C");

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

Diferencias Clave

Característica	UniversalTelegramBot	KissTelegram
Pérdida de mensajes	Sí (si WiFi falla)	No (LittleFS)
Dependencias	ArduinoJson	Ninguna
Gestión de memoria	String (leaks)	char[] (sin leaks)
Prioridades	No	✅ 4 niveles
Power management	No	✅ 6 modos
OTA integrado	No	✅ Dual-bank
Uptime >49 días	❌ Overflow	✅ Gestionado
WiFi inteligente	No	✅ Race-safe

Características Principales

1. Persistencia LittleFS

Todos los mensajes se guardan en flash antes de enviarse:

```
cpp

bot->queueMessage(chat_id, "Alerta temperatura", PRIORITY_CRITICAL);
// ✅ Guardado en /queue_messages.json ANTES de intentar enviar
```

2. Prioridades de Mensajes

```
cpp

PRIORITY_LOW    // Logs, debug
PRIORITY_NORMAL // Mensajes normales (default)
PRIORITY_HIGH   // Alertas
PRIORITY_CRITICAL // Emergencias - activa modo TURBO
```

3. Power Management

6 modos automáticos según actividad:

```
cpp

POWER_BOOT    // Iniciando (30s)
POWER_LOW     // Sin mensajes >5min
POWER_IDLE    // Sin mensajes recientes
POWER_ACTIVE  // Procesando mensajes
POWER_TURBO   // Mensajes CRITICAL
POWER_MAINTENANCE // Durante OTA
```

4. WiFi Manager Inteligente

Detecta calidad de conexión y espera a que sea estable:

```
cpp
```

QUALITY\_DEAD // Sin conexión

QUALITY\_POOR // Conectado pero inestable

QUALITY\_FAIR // Conexión aceptable

QUALITY\_GOOD // Conexión estable >30s

QUALITY\_EXCELLENT // Conexión perfecta

Arquitectura Técnica

char[] vs String: Eliminando Memory Leaks

Problema con String (UniversalTelegramBot y otras libs):

cpp

// ❌ CÓDIGO TÍPICO CON String

void processMessage() {

String message = "Temperatura: ";

message += String(temp);

message += "°C";

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

// ❌ PROBLEMAS:

// 1. Fragmentación del heap

// 2. Memory leaks en concatenaciones

// 3. Reallocs constantes

// 4. Crash tras horas/días de operación

}

// Después de 100 iteraciones:

Serial.printf("Heap: %d bytes\n", ESP.getFreeHeap());

// Heap va BAJANDO gradualmente

// Fragmentación aumenta

// Sistema INESTABLE tras 24-48 horas

Solución KissTelegram con char[]:

cpp

// ✅ CÓDIGO KISSTELEGRAM

void processMessage() {

char message[128];

snprintf(message, sizeof(message), "Temperatura: %.1f°C", temp);

bot->queueMessage(chatId, message);

// ✅ VENTAJAS:

// 1. Stack allocation (liberación automática)

// 2. Zero fragmentación

// 3. Zero memory leaks

// 4. ESTABLE 24/7/365

}

// Después de 1000 iteraciones:

Serial.printf("Heap: %d bytes\n", ESP.getFreeHeap());

// Heap CONSTANTE

// Zero fragmentación

// Sistema ESTABLE indefinidamente

Comparativa Real:

Métrica	String (libs comunes)	char[] (KissTelegram)
Heap tras 1h	~15KB	0KB
Heap tras 24h	~50KB+	0KB
Fragmentación	Alta (fatal)	Nula
Crashes	Frecuentes	Ninguno
Uptime máximo	2-7 días	Indefinido

Implementación Interna KissTelegram:

cpp

```
// system_setup.h - Buffers pre-dimensionados
#define JSON_BUFFER_SIZE 8192
#define MESSAGE_BUFFER_SIZE 2048

class KissTelegram {
private:
    // ✅ Buffers estáticos en heap (una sola vez)
    char* jsonBuffer;
    char* messageBuffer;
    char* commandBuffer;
    char* paramBuffer;

    // Constructor
    KissTelegram(const char* token) {
        // Allocación ÚNICA al inicio
        jsonBuffer = new char[JSON_BUFFER_SIZE];
        messageBuffer = new char[MESSAGE_BUFFER_SIZE];
        commandBuffer = new char[COMMAND_BUFFER_SIZE];
        paramBuffer = new char[PARAM_BUFFER_SIZE];

        // ✅ Nunca más allocs/deallocs
        // ✅ Zero fragmentación
    }

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

#### Recomendaciones para tu código:

```
cpp

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

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

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

#### Gestión de millis() Overflow: Operación 24/7

##### Problema de millis() en producción:

```
cpp

// ❌ CÓDIGO TÍPICO (falla tras 49 días)
unsigned long lastCheck = millis();


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

    // ❌ FALLA cuando millis() hace overflow (49.7 días)
    if(now - lastCheck > 60000) {
        checkSensor();
        lastCheck = now;
    }
}

// Ejemplo del fallo:
// Día 49: lastCheck = 4294967295 (casi overflow)
// Día 50: now = 100 (overflow ocurrió)
// now - lastCheck = 100 - 4294967295 = NEGATIVO ENORME
// ❌ Condición NUNCA se cumple
```


##### Solución KissTelegram:

cpp

```
//  KissTelegram.cpp - Función safe
unsigned long KissTelegram::safeTimeDiff(unsigned long later, unsigned long earlier) {
    // Maneja correctamente el overflow de 32-bit
    return (later >= earlier)
        ? (later - earlier) // Caso normal
        : ((0xFFFFFFFF - earlier) + later + 1); // Caso overflow
}


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

    if(timeSinceLastMsg < minMessageInterval) return;

    //  Funciona SIEMPRE, incluso tras 49+ días
}
```


#### Ejemplo práctico para tu código:

cpp

```
//  COPIA esta función a tu sketch
unsigned long safeTimeDiff(unsigned long later, unsigned long earlier) {
    return (later >= earlier)
        ? (later - earlier)
        : ((0xFFFFFFFF - earlier) + later + 1);
}

// Uso en sensores de campo:
unsigned long lastSensorRead = 0;




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

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

#### Validación del overflow:

cpp

```
// Test simulado (no esperar 49 días)
void testOverflow() {
    unsigned long testCases[][3] = {
        // {earlier, later, expected_diff}
        {100, 200, 100}, // Normal
        {4294967295, 100, 101}, // Overflow
        {4294967000, 1000, 1296}, // Overflow parcial
    };

    for(int i = 0; i <= 3; i++) {
        unsigned long diff = safeTimeDiff(testCases[i][1], testCases[i][0]);
        Serial.printf("Test %d: %lu (esperado %lu) %s\n",
            i, diff, testCases[i][2],
            diff == testCases[i][2] ? "
// Test 1: 101 (esperado 101) 
// Test 2: 1296 (esperado 1296) 
```

#### Uso en KissTelegram internamente:

cpp

```

// Todos los timeouts usan 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(): Evitando Race Conditions

### Problema común - Race Conditions en WiFi:

```

cpp

// ❌ CÓDIGO TÍPICO
void loop() {
    // Check 1: WiFi parece OK
    if(WiFi.status() == WL_CONNECTED) {

        // ❌ PROBLEMA: WiFi puede caerse AQUÍ
        // (entre el check y el connect)

        if(client.connect("api.telegram.org", 443)) {
            client.print("GET /bot...");
            // ❌ Connection failed
            // ❌ Mensaje perdido
            // ❌ CPU malgastada en retries
        }
    }
}

// Síntomas:
// - "Connection failed" en serial
// - Mensajes perdidos
// - CPU al 100% reintentando
// - Watchdog resets

```

### Solución KissTelegram - WiFi Manager Inteligente:

```

cpp

```

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

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

    // 2. Verificar que se inicializó
    if(wifiStableTime == 0) return false;

    // 3. Test de conectividad real
    if(!testBasicConnectivity()) {
        failedPings++;
        currentQuality = QUALITY_POOR;
        return false;
    }

    // 4. Verificar tiempo de estabilidad (30s por defecto)
    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;
}
```

#### Uso en tu código:

```
cpp

void loop() {
    // 🟢 Check inteligente con múltiples validaciones
    if(bot->isWifiStable()) {
        // WiFi REALMENTE estable (30s+ conectado + ping OK)
        bot->processQueue();
        bot->checkMessages(handleCommands);
    } else {
        // WiFi inestable o recién conectado
        // 🟢 NO malgastar CPU en requests que fallarán
        Serial.println(" 🕒 Esperando WiFi estable...");
    }

    delay(1000);
}
```

#### Ventajas del sistema:

```
cpp
```

```
// 1. Evita work innecesario
if(!bot->isWifiStable()) {
    // ✅ NO ejecutar:
    // - HTTP requests
    // - Telegram API calls
    // - Procesamiento de cola
    // → Ahorro de CPU y batería
    return;
}

// 2. Calidad de conexión gradual
switch(bot->getConnectionQuality()) {
    case QUALITY_DEAD:
        // WiFi caído
        break;

    case QUALITY_POOR:
        // Conectado pero fallan pings
        // ✅ Esperar antes de enviar
        break;

    case QUALITY_FAIR:
        // Conexión aceptable
        // ✅ OK para enviar mensajes NORMAL
        break;

    case QUALITY_GOOD:
        // Estable >30s, sin fallos
        // ✅ OK para TODO
        break;
}

// 3. Gestión automática de reconexión
void checkWifiStability() {
    static bool wasConnected = false;
    bool isConnected = (WiFi.status() == WL_CONNECTED);

    if(wasConnected && !isConnected) {
        // WiFi CAÍDO
        bot->disable(); // ✅ Para procesamiento
        Serial.println("🔴 WiFi desconectado");
    }

    if(!wasConnected && isConnected) {
        // WiFi RECUPERADO
        Serial.println("🟢 WiFi recuperado - estabilizando...");
        delay(5000); // Esperar estabilización
        bot->setWifiStable();
        bot->enable(); // ✅ Reactiva procesamiento
    }

    wasConnected = isConnected;
}
```

Comparativa de comportamiento:

Escenario	Código típico	KissTelegram
WiFi cae durante envío	❌ Crash/hang	✅ Detecta y para
Reconexión WiFi	❌ Envío inmediato (falla)	✅ Espera 30s estable
Ping falla	❌ No detecta	✅ Marca QUALITY_POOR
CPU malgastada	❌ Retries constantes	✅ Solo si stable

Power Management en Campo

¿Por qué Power Management en IoT?

Escenario típico - Sensor remoto:

Ubicación: Estación meteorológica en montaña

Alimentación: Panel solar + batería

Conexión: 4G/LTE intermitente

Requisito: Uptime 365 días/año

Problema sin Power Management:

```
cpp
```



```
// ❌ CÓDIGO SIN POWER MANAGEMENT

void loop() {
    // CPU SIEMPRE al 100%
    checkWiFi();           // 30% CPU
    processQueue();         // 40% CPU
    checkMessages();        // 30% CPU

    // Resultado:
    // - Batería dura 2-3 días con panel
    // - Sin luz solar → apagón en 12 horas
    // - WiFi scanning constante → interferencias

    delay(100); // CPU idle solo 10% del tiempo
}

// Consumo típico: 150-200mA promedio
// Batería 5000mAh → ~25 horas sin sol
```

#### Solución KissTelegram - Power Management:

```
cpp

// ✅ CÓDIGO CON POWER MANAGEMENT
void loop() {
    // CPU se adapta a la carga de trabajo
    bot->updatePowerState(); // Evalúa modo necesario

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

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

    // Delay adaptativo según modo
    int delayTime = bot->getRecommendedDelay();
    delay(delayTime); // 500ms - 10000ms según modo

    // Resultado:
    // - Batería dura 7-15 días con panel
    // - Sin luz solar → 3-5 días de autonomía
    // - WiFi scanning solo cuando necesario
}

// Consumo típico: 30-80mA promedio (60% reducción)
// Batería 5000mAh → ~65 horas sin sol
```

#### Modos de Energía Explicados

##### POWER\_BOOT (primeros 30 segundos):

```
cpp

// Características:
// - WiFi: Inicializando
// - Cola: No procesa
// - Check messages: Cada 30s
// - Delay recomendado: 5000ms

// ¿Cuándo?
// - Arrancando tras reset
// - Conectando WiFi
// - Esperando estabilización

// Consumo: ~120mA
```

##### POWER\_LOW (sin actividad >5 min):

```
cpp
```

```
// Características:
// - WiFi: Mantiene conexión
// - Cola: No procesa
// - Check messages: Cada 60s
// - Delay recomendado: 10000ms

// ¿Cuándo?
// - Sin mensajes pendientes
// - Sin comandos recientes
// - Horario nocturno (opcional)

// Consumo: ~30mA
// → Ideal para ahorro nocturno

// Ejemplo configuración:
void setup() {
  bot->setPowerConfig(
    300, // 5 min inactivo → POWER_LOW
    10,  // 10s transición
    30   // 30s boot stable
  );
}
```

#### POWER\_IDLE (sin actividad reciente):

```
cpp

// Características:
// - WiFi: Activo
// - Cola: Procesa cada 3s
// - Check messages: Cada 15s
// - Delay recomendado: 3000ms

// ¿Cuándo?
// - Cola vacía
// - Sin comandos en 1-5 min
// - Waiting for events

// Consumo: ~60mA
// → Balance ahorro/respuesta
```

#### POWER\_ACTIVE (procesando mensajes):

```
cpp

// Características:
// - WiFi: Activo
// - Cola: Procesa cada 1s
// - Check messages: Cada 10s
// - Delay recomendado: 1000ms

// ¿Cuándo?
// - Mensajes en cola
// - Enviando periódicamente
// - Comandos recientes (<1 min)

// Consumo: ~100mA
// → Modo normal operación
```

#### POWER\_TURBO (emergencias):

```
cpp
```

```
// Características:
// - WiFi: Máxima prioridad
// - Cola: Procesa cada 500ms
// - Check messages: Cada 5s
// - Delay recomendado: 500ms

// ¿Cuándo?
// - Mensajes PRIORITY_CRITICAL
// - Alertas de sistema
// - Recuperación de fallo

// Consumo: ~150mA
// → Solo para emergencias

// Ejemplo trigger automático:
bot->queueMessage(chatId, "🔴 ALARMA", PRIORITY_CRITICAL);
// ✅ Activa POWER_TURBO automáticamente
// ✅ Envía INMEDIATAMENTE
// ✅ Vuelve a ACTIVE tras envío
```

**POWER\_MAINTENANCE (durante OTA):**

```
cpp

// Características:
// - WiFi: Máxima estabilidad
// - Cola: Solo mensajes OTA
// - Check messages: Solo comandos OTA
// - Delay recomendado: Variable

// ¿Cuándo?
// - Proceso OTA activo
// - Descargando firmware
// - Validando actualización

// Consumo: ~120mA
// → Necesario para OTA estable

// Activación automática:
bot->setMaintenanceMode(true, "OTA en progreso");
// ✅ Para mensajes normales
// ✅ Permite solo OTA
// ✅ Asegura estabilidad
```

**Configuración para Despliegue en Campo**

**Estación meteorológica autónoma:**

```
cpp
```

```
void setup() {
  // Power management agresivo
  bot->setPowerSaving(true);
  bot->setPowerConfig(
    600, // 10 min idle → POWER_LOW
    15,  // 15s transiciones
    45   // 45s boot stable
  );

  // Modo operación ahorro
  bot->setOperationMode(MODE_POWERSAVE);
  // - MinMessageInterval: 2000ms
  // - WiFi checks reducidos
  // - CPU idle maximizado
}

void loop() {
  // Actualizar cada hora (fuera de emergencias)
  static unsigned long lastReport = 0;
  if(safeTimeDiff(millis(), lastReport) >= 3600000) {
    float temp = readTemp();
    float hum = readHumidity();

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

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

  // Alertas críticas (bypass power management)
  if(temp >= 50.0) {
    bot->queueMessage(chatId,
      "🔥 Temperatura crítica",
      PRIORITY_CRITICAL);
    // ✅ Activa POWER_TURBO
    // ✅ Envío inmediato
  }

  // Gestión inteligente
  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%
  // ✅ Batería dura 5x más
}
```

Sensor agrícola con solar:

cpp

```

void setup() {
    // Balance ahorro/respuesta
    bot->setOperationMode(MODE_BALANCED);

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

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

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

        float soilHumidity = readSoil();

        if(soilHumidity < 20.0) {
            // 🚨 Necesita riego
            bot->queueMessage(chatId,
                " 🚨 Suelo seco - riego necesario",
                PRIORITY_HIGH);
            // ✅ Sube a POWER_ACTIVE para envío rápido
        } else {
            // 📺 Reporte normal
            char msg[100];
            snprintf(msg, sizeof(msg),
                " 📺 Humedad suelo: %.1f%%", soilHumidity);
            bot->queueMessage(chatId, msg, PRIORITY_NORMAL);
            // ✅ Se envía cuando WiFi y batería lo permitan
        }

        lastMeasure = millis();
    }

    // Gestión automática
    bot->updatePowerState();

    // Solo procesar si tiene sentido
    if(bot->shouldProcessQueue()) {
        bot->processQueue();
    }

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

    // Ahorro nocturno adicional (opcional)
    struct tm timeinfo;
    if(getLocalTime(&timeinfo)) {
        if(timeinfo.tm_hour >= 22 || timeinfo.tm_hour <= 6) {
            // Noche: forzar POWER_LOW
            bot->setPowerMode(POWER_LOW);
        }
    }

    delay(bot->getRecommendedDelay());
}

```

## Monitoreo industrial 24/7:

cpp

```
void setup() {
    // Máxima respuesta
    bot->setOperationMode(MODE_PERFORMANCE);

    // Power management mínimo (alimentado por red)
    bot->setPowerSaving(false);
    // ✅ Siempre en POWER_ACTIVE o POWER_TURBO
}

void loop() {
    // Monitoreo continuo
    float pressure = readPressure();
    float flow = readFlow();

    if(pressure > MAX_PRESSURE) {
        // 🚨 Alerta crítica
        bot->queueMessage(chatId,
            " 🚨 PRESIÓN CRÍTICA - Apagado automático",
            PRIORITY_CRITICAL);

        shutdownSystem(); // Seguridad
    }

    // Reporte cada 5 min (aunque no haya power management)
    static unsigned long lastReport = 0;
    if(safeTimeDiff(millis(), lastReport) > 300000) {
        char msg[200];
        snprintf(msg, sizeof(msg),
            " 🏭 ESTADO PLANTA\n"
            " ⚙️ Presión: %.1f bar\n"
            " 💧 Caudal: %.1f L/min\n"
            " ⌚ Uptime: %lu h",
            pressure, flow,
            millis() / 3600000);

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

    // Sin power management → siempre procesa
    bot->processQueue();
    bot->checkMessages(handleCommands);

    delay(500); // Respuesta rápida
}
```

Estadísticas de Power Management

cpp

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

    const char* nombres[] = {
        "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",
            nombres[i],
            timeInMode / 60000);
    }

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

    // Ejemplo output tras 24h:
    // ⚡ ESTADÍSTICAS POWER:
    // - BOOT: 1 min
    // - LOW: 720 min (12h nocturnas)
    // - IDLE: 500 min
    // - ACTIVE: 218 min
    // - TURBO: 1 min
    // - MAINT: 0 min
    //
    // 📊 Eficiencia: 0.08 msg/s
    // ✅ 50% del tiempo en LOW/IDLE → 60% ahorro energía
}

// Llamar cada 6 horas:
static unsigned long lastStats = 0;
if(safeTimeDiff(millis(), lastStats) > 21600000) {
    printPowerStats();
    bot->resetPowerStatistics(); // Reset contadores
    lastStats = millis();
}
```

Instalación y Configuración

Paso 1: Hardware Requerido

- ESP32 con PSRAM (mínimo 8MB)
- Flash mínimo 4MB (recomendado 16MB)

Paso 2: Partition Table

Editar `boards.txt` de tu ESP32 para incluir:

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

Paso 3: Configurar Credenciales

Editar `system_setup.h`:

```
cpp

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

Paso 4: Código Básico

```
cpp
```

```

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

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

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

  // 1. Inicializar credenciales
  creds.begin();

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

  // 3. Crear bot y restaurar mensajes
  bot = new KissTelegram(creds.getBotToken());
  bot->restoreFromStorage(); // ✅ Recupera mensajes pendientes

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

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

void loop() {
  bot->processQueue(); // Envía mensajes pendientes
  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 activo");
  }
}

```

## Persistencia de Mensajes - Cero Pérdidas

### Escenario 1: Corte de WiFi

#### Problema común (UniversalTelegramBot):

```

cpp

// ❌ Si WiFi cae aquí, mensaje perdido
bot.sendMessage(CHAT_ID, "Alerta sensor", "");

```

#### Solución KissTelegram:

```

cpp

// ✅ Mensaje guardado en LittleFS inmediatamente
bot->queueMessage(chat_id, "Alerta sensor", PRIORITY_HIGH);

// WiFi se cae...
// ✅ Mensaje sigue en /queue_messages.json

// WiFi vuelve...
bot->processQueue(); // ✅ Envía automáticamente

```

### Escenario 2: Reinicio Inesperado

#### Simulación:

```

cpp

```



```
void loop() {
  // Usuario envía 10 mensajes
  for(int i = 0; i < 10; i++) {
    char msg[50];
    snprintf(msg, sizeof(msg), "Mensaje %d", i);
    bot->queueMessage(chat_id, msg);
  }

  // ❌ ESP32 se reinicia AQUÍ (corte de corriente)
  ESP.restart();
}

// Tras reiniciar:
void setup() {
  bot = new KissTelegram(BOT_TOKEN);
  bot->restoreFromStorage();
  // ✅ Los 10 mensajes siguen ahí
  Serial.printf("Mensajes pendientes: %d\n", bot->getMessagesInFS());
}
```

Escenario 3: Actualización OTA

```
cpp

// Usuario tiene 50 mensajes pendientes en LittleFS
Serial.printf("Pendientes antes OTA: %d\n", bot->getMessagesInFS());

// Inicia OTA con /ota
// Durante OTA → maintenance mode activo
// ✅ No se pierden mensajes

// Tras actualizar:
bot->restoreFromStorage();
Serial.printf("Pendientes tras OTA: %d\n", bot->getMessagesInFS());
// ✅ Siguen los 50 mensajes
```

Verificar Persistencia

Test manual:

```
cpp

void testPersistencia() {
  // 1. Encolar 20 mensajes
  for(int i = 0; i < 20; i++) {
    char msg[50];
    snprintf(msg, sizeof(msg), "Test persistencia %d", i);
    bot->queueMessage(chat_id, msg);
  }

  // 2. Ver estado
  Serial.printf("En FS: %d\n", bot->getMessagesInFS());

  // 3. Reiniciar manualmente
  delay(2000);
  ESP.restart();

  // 4. En setup() tras reinicio:
  bot->restoreFromStorage();
  Serial.printf("Recuperados: %d\n", bot->getMessagesInFS());
  // ✅ Debe mostrar 20
}
```

Comandos útiles:

```
/debugfs → Ver contenido de LittleFS
/storage → Estadísticas de almacenamiento
/cleanup → Limpiar mensajes enviados
/borrar confirmar → Borrar todo
```

Gestión de Prioridades

Niveles de Prioridad

```
cpp
```

```
// 🟢 LOW - Logs, debug (no activa power management)
bot->queueMessage(chat_id, "Debug: Variable X = 42", PRIORITY_LOW);

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

// 🟡 HIGH - Alertas
bot->queueMessage(chat_id, " ⚠ Temperatura alta", PRIORITY_HIGH);

// 🔴 CRITICAL - Emergencias (activa modo TURBO)
bot->queueMessage(chat_id, " 🚨 FALLO CRÍTICO", PRIORITY_CRITICAL);
```

Ejemplo: Sensor con Alertas

```
cpp

void monitorearTemperatura() {
    float temp = leerSensor();

    if(temp > 80) {
        // 🔴 Emergencia → salta toda la cola
        char msg[100];
        snprintf(msg, sizeof(msg), " 🚨 TEMPERATURA CRÍTICA: %.1f°C", temp);
        bot->queueMessage(chat_id, msg, PRIORITY_CRITICAL);

    } else if(temp > 60) {
        // 🟡 Alerta → prioridad sobre mensajes normales
        char msg[100];
        snprintf(msg, sizeof(msg), " ⚠ Temperatura alta: %.1f°C", temp);
        bot->queueMessage(chat_id, msg, PRIORITY_HIGH);

    } else {
        // 🔵 Info normal
        char msg[100];
        snprintf(msg, sizeof(msg), " 📊 Temperatura: %.1f°C", temp);
        bot->queueMessage(chat_id, msg);
    }
}
```

Test de Prioridades

Comando `(/prioridades)` ejecuta:

```
cpp

void testPrioridades() {
    // Encolar en orden inverso a prioridad
    bot->queueMessage(chat_id, " 🟢 Mensaje LOW", PRIORITY_LOW);
    bot->queueMessage(chat_id, " 🔵 Mensaje NORMAL", PRIORITY_NORMAL);
    bot->queueMessage(chat_id, " 🟡 Mensaje HIGH", PRIORITY_HIGH);
    bot->queueMessage(chat_id, " 🔴 Mensaje CRITICAL", PRIORITY_CRITICAL);

    // ✅ Se envían en orden: CRITICAL ⇒ HIGH ⇒ NORMAL ⇒ LOW
}
```

Comandos del Sistema

Comandos Básicos

```
/help o /start
└─ Muestra ayuda completa

/estado
└─ Estado del sistema:
    • Uptime
    • Mensajes enviados/pendientes
    • Memoria libre
    • Modo energía
    • Auto-mensajes ON/OFF

/config
└─ Configuración al monitor serie

/stats
└─ Estadísticas acumuladas (lifetime):
    • Total encolados
    • Total enviados
```

- Tasa de éxito
- Memoria

### Comandos de Diagnóstico

```
/debug
└─ Info detallada al serial

/debugfs
└─ Análisis de LittleFS:
    • Mensajes pendientes (s:0)
    • Mensajes enviados (s:1)
    • Muestra JSON en serial

/memoria
└─ Info de memoria:
    • Heap libre
    • Heap mínimo
    • LittleFS usado

/storage
└─ Estado de almacenamiento
```

### Comandos de Gestión

```
/cleanup
└─ Limpiar mensajes enviados del FS

/llevar [N]
└─ Test: encolar N mensajes (default 15)

/forceprocess
└─ Forzar envío de cola

/borrar confirmar
└─ ⚠ Eliminar TODOS los mensajes pendientes

/prioridades
└─ Test de sistema de prioridades

/testlittlefs
└─ Test de persistencia (genera 10 msg)
```

### Comandos de Sistema

```
/parar
└─ Desactivar mensajes automáticos

/activar
└─ Reactivar mensajes automáticos

/changepin <viejo> <nuevo>
└─ Cambiar PIN OTA

/changebuk <viejo> <nuevo>
└─ Cambiar PUK OTA

/credentials
└─ Ver credenciales en serial

/resetcredentials CONFIRMAR
└─ ⚠ Reset a valores por defecto
```

## Actualización OTA

### Proceso Completo

#### 1. Iniciar OTA

Usuario: /ota  
Bot: 📡 ACTUALIZACIÓN OTA

- Durante la actualización:
- Los mensajes quedan almacenados
  - No perderá mensajes
  - No recibirá mensajes

Escriba /otapin [PIN] para iniciar  
🕒 Timeout: 60 segundos

## 2. Autenticar con PIN

Usuario: /otapin 1234  
Bot: ✅ PIN CORRECTO

- Iniciando proceso...
- Mensajes pausados
  - Verificando espacio
  - Creando backup

🕒 Timeout proceso: 7 minutos

## 3. Enviar Firmware

- Enviar archivo (.bin) por Telegram
- Bot descarga a PSRAM y verifica checksum

Bot: 📡 DESCARGANDO FIRMWARE

Recibiendo: firmware\_v2.bin  
Tamaño: 1.2 MB

No apague el dispositivo.

## 4. Confirmar Flash

Bot: ✅ FIRMWARE VERIFICADO

Archivo: firmware\_v2.bin  
Tamaño: 1.2 MB  
CRC32: 0x12345678

⚠️ CONFIRMAR FLASH

/otaconfirm - Flashear ahora  
/otacancel - Cancelar proceso

🕒 Timeout: 3 minutos

Usuario: /otaconfirm

Bot: 🔥 FLASHEANDO FIRMWARE

Escribiendo en memoria flash...  
⚠️ NO APAGUE EL DISPOSITIVO

## 5. Validar tras Reinicio

[ESP32 reinicia automáticamente]

Bot:  VALIDACIÓN FIRMWARE

El dispositivo ha arrancado.

 Tiene 60 segundos para validar

/otaok - Confirmar que funciona

/otacancel - Forzar rollback

Si no responde se ejecutará rollback automático.

Usuario: /otaok

Bot:  FIRMWARE VALIDADO

La actualización se completó exitosamente.

Backup eliminado.

Sistema operativo en nuevo firmware.

(Puede ignorar el mensaje anterior de validación)

### Rollback Automático

Si NO envías /otaok en 60 segundos:

[Pasan 60 segundos sin /otaok]

Bot:  TIMEOUT VALIDACIÓN

No se recibió confirmación en 60 segundos.

Iniciando rollback automático...

El dispositivo se reiniciará con firmware anterior.

[ESP32 reinicia con firmware original]

Bot:  Rollback completado

Sistema operativo en firmware anterior

### Rollback Manual

Si detectas problemas tras actualizar:

Usuario: /otacancel

Bot:  INICIANDO ROLLBACK

Restaurando firmware anterior...

El dispositivo se reiniciará.

No apague el dispositivo.

[ESP32 reinicia con firmware original]

### Seguridad OTA

**Bloqueo por PIN incorrecto:**

Usuario: /otapin 0000

Bot:  PIN incorrecto

Intentos restantes: 2

Usuario: /otapin 1111

Bot:  PIN incorrecto

Intentos restantes: 1

Usuario: /otapin 2222

Bot:  PIN BLOQUEADO

Ha superado el máximo de 3 intentos.

Para desbloquear escriba /otapuk [PUK]

**Desbloqueo con PUK:**

Usuario: /otapuk 12345678  
Bot: 🤖 OTA DESBLOQUEADO

El PIN ha sido desbloqueado.  
Los intentos se han reiniciado.

Para iniciar actualización escriba /ota

## Ejemplos Prácticos

### Ejemplo 1: Sensor de Temperatura (Optimizado para Campo)

```
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 para campo
    bot->setPowerSaving(true);
    bot->setOperationMode(MODE_POWERSAVE);
}

void loop() {
    // ✅ Solo procesar si WiFi realmente estable
    if(bot->isWifiStable()) {
        bot->processQueue();
    }

    // Leer sensor cada 5 minutos
    static unsigned long lastRead = 0;
    unsigned long now = millis();

    // ✅ Usar safeTimeDiff para uptime > 49 dias
    if(safeTimeDiff(now, lastRead) > 300000) {
        float temp = leerTemperatura();

        // ✅ char[] en vez de String
        char msg[100];
        sprintf(msg, sizeof(msg), "🌡 Temperatura: %.1f°C", temp);

        // ✅ Si WiFi falla, mensaje guardado en LittleFS
        bot->queueMessage(creds.getChatId(), msg);

        lastRead = now;
    }

    // ✅ Delay adaptativo según power mode
    delay(bot->getRecommendedDelay());
}

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

### Ejemplo 2: Sistema de Alertas (Sin Memory Leaks)

cpp

```

void monitorearSistema() {
    // Leer sensores
    float temp = leerTemperatura();
    float humedad = leerHumedad();
    int bateria = leerBateria();

    // ✅ Buffers char[] pre-dimensionados
    char msg[200];

    // Temperatura crítica
    if(temp > 80) {
        snprintf(msg, sizeof(msg),
            "🔥 ALERTA CRÍTICA\n\n"
            "Temperatura: %.1f°C\n"
            "Batería: %d%%\n"
            "Acción requerida inmediatamente",
            temp, bateria);
        bot->queueMessage(creds.getChatId(), msg, PRIORITY_CRITICAL);
        // ✅ Activa POWER_TURBO automáticamente
    }

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

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

    // Reporte normal cada hora
    static unsigned long lastReport = 0;
    unsigned long now = millis();

    // ✅ safeTimeDiff para uptime > 49 días
    if(safeTimeDiff(now, lastReport) > 3600000) {
        snprintf(msg, sizeof(msg),
            "📊 REPORTE HORARIO\n\n"
            "🌡️ Temp: %.1f°C\n"
            "💧 Humedad: %.1f%%\n"
            "🔋 Batería: %d%%\n"
            "🕒 Uptime: %lu min",
            temp, humedad, bateria,
            now / 60000);
        bot->queueMessage(creds.getChatId(), msg, PRIORITY_NORMAL);
        lastReport = now;
    }
}

// ✅ Zero memory leaks
// ✅ Heap constante tras 1000+ iteraciones

```

### Ejemplo 3: Control Remoto (WiFi Inteligente)

cpp

```

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

    // ✅ Buffers char[] para respuestas
    char response[300];

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

    // Leer sensor bajo demanda
    else if(strcmp(command, "/temp") == 0) {
        float temp = leerTemperatura();
        sprintf(response, sizeof(response),
            "🌡 Temperatura actual: %.1f°C", temp);
        bot->sendMessage(chat_id, response);
    }

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

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

        sprintf(response, sizeof(response),
            "📊 ESTADO DEL SISTEMA\n\n"
            "🕒 Uptime: %lu min\n"
            "💾 Heap libre: %d bytes\n"
            "📬 Mensajes pendientes: %d\n"
            "🔌 Power Mode: %d\n"
            "📶 WiFi: %s",
            uptime / 60000,
            ESP.getFreeHeap(),
            bot->getMessagesInFS(),
            bot->getCurrentPowerMode(),
            bot->isWifiStable() ? "Conectado" : "Desconectado");
        bot->sendMessage(chat_id, response);
    }
}

void loop() {
    // ✅ Solo procesar si WiFi REALMENTE estable
    // Evita race conditions y trabajo innecesario
    if(bot->isWifiStable()) {
        bot->processQueue();
        bot->checkMessages(handleCommands);
    } else {
        Serial.println("⌚ Esperando WiFi estable...");
    }

    delay(1000);
}

```

#### Ejemplo 4: Logger con Persistencia (Producción 24/7)

cpp



```

// Sistema de logging que nunca pierde mensajes
// 🟢 Uptime >49 días sin problemas

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);
    }
};

// Uso:
Logger logger;
unsigned long bootTime;

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

    logger.info("Sistema iniciado");
    logger.debug("Configurando sensores...");

    if(!inicializarSensores()) {
        logger.error("Fallo al inicializar sensores");
    }
}




void loop() {
    if(detectarError()) {
        logger.warning("Error detectado en sensor X");
    }
}

// 🟢 Todos los logs guardados en LittleFS
// 🟢 Se envían automáticamente cuando WiFi esté estable
// 🟢 Sobreviven a reinicios

// Estadísticas cada 24h
static unsigned long lastStats = 0;
unsigned long now = millis();

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

```

```
}  
}  
  
//  Zero memory leaks  
//  Funciona >49 días sin overflow  
//  WiFi inteligente evita race conditions
```

## Troubleshooting

**Problema:** Mensajes no se envían



**Diagnóstico:**

```
/estado    → Ver mensajes pendientes  
/debugfs   → Ver contenido de LittleFS  
/debug      → Info completa al serial
```

**Posibles causas:**

1. WiFi no estable → esperar `bot->setWifiStable()`
2. Maintenance mode activo → verificar con `/estado`
3. Power mode bajo → mensajes se envían más lento

**Solución:**

```
cpp  
  
// Forzar procesamiento  
bot->processQueue();  
  
// Verificar estado WiFi  
if(bot->isWifiStable()) {  
    Serial.println( WiFi OK");  
} else {  
    Serial.println( WiFi inestable");  
}
```

**Problema:** "Cola llena"

**Diagnóstico:**

```
/storage    → Ver uso de LittleFS  
/debugfs    → Contar mensajes pendientes
```

**Solución:**

```
/cleanup     → Limpiar enviados  
/borrar confirmar → Borrar todo (emergencia)
```

**Problema:** Memoria baja

**Diagnóstico:**

```
/memoria     → Ver heap disponible
```

**Solución:**

```
cpp  
  
// Limpiar mensajes enviados más frecuentemente  
bot->cleanupStorage();  
  
// Reducir intervalo de auto-save  
bot->setAutoSaveInterval(60000); // 1 minuto
```

**Problema:** Sistema inestable tras horas

**Diagnóstico:**

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

**Causa probable:** Memory leaks por uso de String

**Solución:**

```
cpp
```

```
// ❌ NO USAR
String msg = "Temp: " + String(temp);

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

**Problema:** Crashes tras 49 días

**Causa:** Overflow de millis() no gestionado

**Solución:**

```
cpp

// ✅ Usar safeTimeDiff en TODO el código
unsigned long safeTimeDiff(unsigned long later, unsigned long earlier) {
    return (later >= earlier)
        ? (later - earlier)
        : ((0xFFFFFFFF - earlier) + later + 1);
}
```

**Problema:** OTA no inicia

**Verificar:**

- 1. PIN correcto
- 2. Espacio en LittleFS ((debugfs))
- 3. PSRAM disponible

**Logs importantes:**

 Solicitud OTA recibida...

 Espacio suficiente

 Iniciando backup...

**Problema:** Rollback no funciona

**Verificar en serial:**

 Validando firmware nuevo...

 Tiene 60 segundos para validar

Si no aparece → revisar (isFirstBootAfterOTA()) en código.

## Configuración Avanzada

### Power Management

```
cpp

// Personalizar timeouts
bot->setPowerConfig(
    300, // idleTimeout: 5 min sin mensajes → POWER_LOW
    10, // decayTime: 10s para transiciones
    30 // bootStableTime: 30s tras boot
);

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

### Callbacks

```
cpp

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

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

### Configuración de Storage

```
cpp
```

```
// Intervalo de auto-guardado
bot->setAutoSaveInterval(300000); // 5 minutos

// Máximo de mensajes en storage
bot->setMaxQueueStorage(500);

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

Conclusión

KissTelegram está diseñado para **despliegues IoT críticos** donde la pérdida de mensajes no es una opción. Su arquitectura basada en LittleFS garantiza que cada mensaje llegue a destino, sin importar:

- Calidad de WiFi
- Cortes de energía
- Reinicios del sistema
- Actualizaciones OTA

Características únicas:

- ☒ Cero dependencias externas
- ☒ Cero memory leaks (char[] puro)
- ☒ Cero pérdida de mensajes
- ☒ OTA dual-bank seguro
- ☒ Power management inteligente
- ☒ Uptime >49 días (gestión millis())
- ☒ WiFi race-safe

Ventajas Técnicas Críticas:

Aspecto	Implementación	Beneficio
Memoria	char[] vs String	Zero leaks, uptime indefinido
Tiempo	safeTimeDiff()	Funciona >49 días sin crash
WiFi	isWifiStable()	Evita race conditions y CPU waste
Energía	6 power modes	60% ahorro batería en campo
Persistencia	LittleFS primario	Cero mensajes perdidos

¿Preguntas o problemas?

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

Versión del Manual: 5.0.0  
Fecha: Noviembre 2024  
Autor: Vicente Soriano