FIELDCRAFT DIGITAL

BMS Firmware Diagnostic Guide

S32K144 Flash & RAM Analysis, Looptime Mapping, and Production Test Architecture

DIAGNOSE. OPTIMIZE. DEPLOY.

Edition v1.0 | 2026 | $19.00

Table of Contents

Chapter 1: Introduction

1.1 Purpose and Scope

This guide provides a comprehensive diagnostic reference for Battery Management System (BMS) firmware engineers working with the NXP S32K144 microcontroller. The document covers memory analysis, real-time performance profiling, production testing, and field diagnostic procedures essential for robust BMS firmware development and validation.

BMS firmware operates in safety-critical environments where failures can result in catastrophic battery pack damage, thermal runaway, or safety hazards. This guide equips engineers with systematic diagnostic approaches to identify, isolate, and resolve firmware-level issues across the complete product lifecycle -- from initial development through production testing and field deployment.

> **Diagnostic Scope**
> *The diagnostic procedures described in this guide address firmware-level analysis only. Hardware fault diagnosis (PCB shorts, component failures, connector issues) requires separate troubleshooting procedures not covered herein.*

1.2 Target Hardware Platform

The primary target platform for this guide is the **NXP S32K144EVB-Q100** evaluation board, featuring the S32K144 microcontroller based on the ARM Cortex-M4F core. Key specifications relevant to diagnostic analysis include:

Table 1.1: S32K144 Key Specifications

Parameter	Value	Diagnostic Relevance
Core	ARM Cortex-M4F @ 80 MHz	WCET baseline, looptime calculation
Flash Memory	512 KB (program + data)	Code size budget, config storage
SRAM	64 KB (SRAM_L + SRAM_U)	Stack sizing, heap allocation
ADC	12-bit SAR, up to 32 channels	Sampling timing, resolution
CAN	3x FlexCAN modules	Message timing, bus loading
GPIO	Up to 128 pins	Pin mapping, signal routing

1.3 Software Stack Overview

The reference BMS firmware stack is built upon FreeRTOS as the real-time operating system kernel, with custom application layers implementing cell voltage monitoring, temperature sensing, State-of-Charge (SOC) estimation, passive cell balancing, and CAN-based diagnostic communication following UDS (Unified Diagnostic Services, ISO 14229).

Table 1.2: Software Stack Components

Layer	Component	Version	Function
RTOS	FreeRTOS	10.4.x	Task scheduling, IPC, timing
HAL	S32 SDK	3.0.x	Driver abstraction
Comm	UDS/ISO-TP	Custom	Diagnostic services
App	BMS Core	Custom	Safety, SOC, balancing
Tools	Segger SystemView	3.50+	Runtime profiling

Chapter 2: S32K144 Flash Memory Analysis

2.1 Flash Memory Map

The S32K144 provides 512 KB of on-chip Flash memory organized as two separate physical arrays: a 256 KB primary program flash array and a 256 KB data flash array. Understanding the flash memory map is critical for proper linker script configuration, bootloader implementation, and over-the-air (OTA) update capability.

Table 2.1: Flash Memory Map Overview

Region	Start Address	Size	Purpose
Bootloader	0x0000_0000	32 KB	Startup, OTA dispatcher
Application	0x0000_8000	384 KB	Main firmware image
Configuration	0x0006_8000	64 KB	NV parameters, DTC storage
OTA Download	0x0007_8000	32 KB	Scratch area for firmware updates

2.2 Sector Breakdown and Allocation

The S32K144 flash is organized into sectors of varying sizes. The program flash uses 4 KB sectors in the lower address range and 8 KB sectors at higher addresses. The data flash uses uniform 2 KB sectors. When designing the flash allocation, sector boundaries must be respected to avoid erase operations affecting adjacent regions.

The **Bootloader** region occupies the first 8 sectors (4 KB each) of program flash. This region contains the vector table, startup code, and the OTA update dispatcher. The bootloader never changes during normal operation and is protected by flash security settings.

The **Application** region is the largest contiguous block, consuming 384 KB. This region holds the main BMS firmware including the RTOS kernel, HAL drivers, application logic, and constant data tables. The application image is checksummed using CRC-32, verified by the bootloader on every startup.

The **Configuration** region stores non-volatile parameters including cell calibration coefficients, contactor cycle counters, DTC records with timestamps, and user-configurable thresholds. This region experiences the most write cycles and requires careful wear leveling.

Table 2.2: Flash Budget Template (Code Size Tracking)

Module	Code (KB)	RO Data (KB)	Total (KB)	Budget %

FreeRTOS Kernel	18.5	2.1	20.6	5.4
S32 SDK HAL	42.0	8.5	50.5	13.1
ADC Driver + ISRs	8.2	1.2	9.4	2.4
FlexCAN Stack	15.8	3.0	18.8	4.9
UDS Services	22.4	4.8	27.2	7.1
BMS Safety Logic	28.6	5.5	34.1	8.9
SOC Estimation (EKF)	35.2	12.0	47.2	12.3
Cell Balancing	12.4	2.8	15.2	3.9
Flash/EEPROM Emulation	14.0	1.5	15.5	4.0
DTC Management	18.8	6.2	25.0	6.5
Application + Utils	85.0	25.5	110.5	28.7
Reserved	--	--	0.4	0.1
TOTAL	**300.9**	**73.1**	**374.4**	**97.5**

2.3 Flash Wear Leveling

Flash memory has a limited erase/program cycle count, typically 100,000 cycles per sector for the S32K144. The configuration region, which stores frequently-updated parameters such as DTC records and cycle counters, requires a wear-leveling algorithm to distribute writes across multiple sectors and extend overall flash lifetime.

The recommended approach for BMS applications is a **ring-buffer with page-level wear tracking**. Each logical record is written to the next available flash page. When the sector fills, the oldest sector is erased and becomes the new tail. A header record on each page tracks the write sequence number, enabling reconstruction of the logical record order after a reset.

```
/* Flash wear-leveling header structure */
typedef struct {
    uint32_t sequenceNumber;    /* Monotonic incrementing counter */
    uint16_t recordType;        /* DTC, CAL, COUNTER, etc. */
    uint16_t dataLength;        /* Payload length in bytes */
    uint32_t timestamp;         /* RTC epoch seconds */
    uint32_t crc32;             /* CRC over header + payload */
} FlashRecordHeader_t;

/* Sector status tracking */
typedef struct {
    uint32_t eraseCount;        /* Cumulative erase cycles */
    uint8_t  status;            /* EMPTY, ACTIVE, FULL, ERASED */
    uint16_t recordsCount;      /* Valid records in sector */
} SectorStatus_t;
```

Design Note

At 100,000 erase cycles per sector and one DTC write every 10 seconds, a 64 KB configuration region with 16 sectors provides approximately 30 years of flash endurance -- well beyond the typical 15-year vehicle service life.

2.4 ECC Handling

The S32K144 flash controller includes single-error-correction, double-error-detection (SECDED) ECC hardware operating on 64-bit flash words. The ECC is calculated over each flash write and verified on every read. Single-bit errors are automatically corrected; double-bit errors trigger a flash access exception.

BMS firmware must handle ECC events as potential indicators of flash degradation or radiation-induced soft errors. The recommended handling strategy involves:

1. Registering a fault exception handler for the Flash Controller Module (FTFC)
2. Logging single-bit corrections via the DTC storage system (DTC P0A0F)
3. Declaring a critical fault on double-bit detection, forcing safe state
4. Periodic scrubbing of the configuration region to correct single-bit errors before they accumulate

```
/* ECC fault handler for flash double-bit errors */
void FTFC_Fault_IRQHandler(void) {
    uint32_t fstat = FTFC->FSTAT;

    if(fstat & FTFC_FSTAT_DFDIF_MASK) {
        /* Double-bit fault detected */
        DTC_SetHardFault(DTC_FLASH_ECC_DOUBLE_BIT);
        BMS_EnterSafeState(SAFE_STATE_REASON_FLASH_FAULT);
        FTFC->FSTAT = FTFC_FSTAT_DFDIF_MASK; /* Clear flag */
    }

    if(fstat & FTFC_FSTAT_SFDIF_MASK) {
        /* Single-bit fault - corrected by HW */
        DTC_SetSoftFault(DTC_FLASH_ECC_SINGLE_BIT);
        FTFC->FSTAT = FTFC_FSTAT_SFDIF_MASK; /* Clear flag */
    }
}
```

2.5 Flash Programming Best Practices

Reliable flash programming in a BMS context requires adherence to several key practices. Flash writes must never occur from code executing in the same flash bank being programmed. The S32K144 supports RWW (Read-While-Write) between program flash and data flash, enabling firmware to execute from program flash while updating configuration records in data flash.

Table 2.3: Flash Programming Best Practices

Practice	Implementation	Rationale
Write from RAM	Copy flash functions to SRAM before execution	Prevents code fetch during erase
Atomic updates	Write header only after payload verified	Prevents corrupted records
Margin reads	Use normal + margin read levels	Detects weakly-programmed bits
Pre-erasure check	Verify sector is non-blank before erase	Saves erase cycles
Dual-bank OTA	Maintain two complete application images	Rollback on corrupt update

Chapter 3: RAM Analysis & Optimization

3.1 RAM Memory Map

The S32K144 provides 64 KB of SRAM divided into two physical regions: **SRAM_L** (0x1FFF_8000 -- 0x1FFF_FFFF, 32 KB) and **SRAM_U** (0x2000_0000 -- 0x2000_7FFF, 32 KB). The split is a legacy of ARM Cortex-M4 architecture where bit-band alias regions map to SRAM_U. For BMS applications, this division must be considered during linker script configuration.

Table 3.1: RAM Allocation Map

Region	Size (KB)	Location	Content
ISR Stack	4	SRAM_L base	Main stack (MSP) for ISRs
Task Stacks	20	SRAM_L + SRAM_U	RTOS task private stacks
FreeRTOS Heap	24	SRAM_U	Dynamic allocation (heap_4)
.data / .bss	12	SRAM_L	Global/static variables
NVIC + Sys	2	Fixed	System vectors, control
Reserved	2	--	Link-time alignment

3.2 Stack Allocation Per Task

Proper stack sizing is critical for BMS firmware safety. An undetected stack overflow can corrupt adjacent memory regions, leading to unpredictable behavior in safety-critical tasks. The recommended approach combines static analysis with runtime watermark monitoring.

Stack size calculation follows the formula:

```
Required Stack = MAX(call_depth_bytes, interrupt_nesting_bytes)
               + local_variables_peak
               + context_save_frame
               + safety_margin(20%)

Context Save Frame = 16 x 4 bytes (R0-R3,R12,LR,PC,xPSR)
                   + 18 x 4 bytes (FPU lazy stacking)
                   + 8 bytes (alignment)
                   = 136 bytes max (with FPU)
```

Table 3.2: Task Stack Allocation

Task	Priority	Allocated (words)	High Water (words)	Utilization %

BMS_Safety	Highest	256	198	77.3
BMS_AdcRead	High	192	145	75.5
BMS_SocCalc	Medium	512	380	74.2
BMS_CanTx	High	128	89	69.5
BMS_Balance	Low	256	112	43.8
BMS_Logger	Lowest	384	201	52.3

The **safety margin** of 20% accounts for worst-case interrupt nesting during task execution. The BMS_Safety task, which executes every 1 ms and handles fault detection, requires the largest margin due to the possibility of multiple nested interrupts (ADC conversion complete + CAN Rx + SysTick).

3.3 Heap Usage Monitoring

FreeRTOS provides several heap allocation implementations (heap_1 through heap_5). For BMS applications, **heap_4** is recommended because it supports dynamic allocation with coalescing of adjacent free blocks, reducing fragmentation over time. Heap_5 extends heap_4 by allowing the heap to span multiple non-contiguous memory regions, which is useful when combining SRAM_L and SRAM_U.

The following code implements a comprehensive RAM usage monitoring function that reports heap statistics and per-task stack high-water marks:

```
/* BMS RAM Usage Monitor - S32K144 */
#include "FreeRTOS.h"
#include "task.h"
#include "stdio.h"

#define RAM_MONITOR_INTERVAL_MS  10000

typedef struct {
    size_t heapFree;
    size_t heapMin;
    size_t totalStackAllocated;
    size_t totalStackFree;
    uint8_t taskCount;
} RamStats_t;

void BMS_PrintRamStats(void) {
    /* Heap usage via FreeRTOS API */
    size_t heapFree = xPortGetFreeHeapSize();
    size_t heapMin  = xPortGetMinimumEverFreeHeapSize();

    /* Task stack high water marks */
    TaskStatus_t *pxTaskStatusArray;
    volatile UBaseType_t uxArraySize, x;

    uxArraySize = uxTaskGetNumberOfTasks();
    pxTaskStatusArray = pvPortMalloc(uxArraySize * sizeof(TaskStatus_t));

    if(pxTaskStatusArray != NULL) {
        uxArraySize = uxTaskGetSystemState(pxTaskStatusArray,
                                           uxArraySize, NULL);

        printf("\n=== BMS RAM Usage Report ===\n");
        printf("Tasks: %u | Heap Free: %u | Min Ever: %u\n\n",
               (unsigned)uxArraySize,
               (unsigned)heapFree,
               (unsigned)heapMin);

        for(x = 0; x < uxArraySize; x++) {
            uint32_t used = pxTaskStatusArray[x].usStackHighWaterMark;
            float pct = (1.0f - (float)used /
                        pxTaskStatusArray[x].uxCurrentPriority) * 100;

            printf("  Task: %-12s | Stack Free: %4u words | "
                   "Load: %5.1f%%\n",
                   pxTaskStatusArray[x].pcTaskName,
                   (unsigned)used, pct);
        }

        vPortFree(pxTaskStatusArray);
    }

    /* Fragmentation estimate */
    size_t heapTotal = configTOTAL_HEAP_SIZE;
    float fragPct = (1.0f - (float)heapFree / heapTotal) * 100.0f;
```

> **Implementation Note**
>
> The `vTaskRamMonitor` task should run at the lowest priority level to avoid interfering with safety-critical operations. Report generation via semihosting or RTT adds latency; for production builds, consider logging to a circular buffer in RAM for deferred retrieval via diagnostic services.

3.4 Memory Fragmentation Analysis

Memory fragmentation occurs when allocated and free blocks become interleaved, preventing the allocation of large contiguous regions even when sufficient total free memory exists. In long-running BMS firmware, fragmentation can eventually cause heap allocation failures.

Fragmentation is quantified using the **fragmentation ratio**:

```
Fragmentation Ratio = 1 - (Largest Free Block / Total Free Memory)
```

A ratio above 0.3 (30%) indicates concerning fragmentation. Mitigation strategies include:

- **Fixed-size pools**: Allocate common structures (CAN frames, DTC records, ADC samples) from pre-allocated pools rather than the general heap
- **Static allocation**: Pre-allocate all buffers at startup; avoid runtime `malloc()` after initialization
- **Block coalescing**: Use heap_4 or heap_5 which automatically merge adjacent free blocks
- **Periodic compaction**: Defragment at known safe points (e.g., during key-off power mode)

Table 3.3: Memory Pool Allocation Strategy

Pool Name	Block Size	Block Count	Total (KB)	Usage
CAN_FramePool	16 bytes	64	1.0	Tx/Rx message buffers
ADC_SamplePool	24 bytes	128	3.0	Cell voltage samples
DTC_RecordPool	32 bytes	32	1.0	Diagnostic trouble codes
LogEntryPool	48 bytes	256	12.0	Flash log ring buffer
EventPool	12 bytes	64	0.75	RTOS event buffers

3.5 RAM Overflow Detection and Prevention

RAM overflow detection requires both compile-time and runtime mechanisms. At compile time, the linker script should define symbols for stack top and heap boundaries, allowing the application to check pointers against these limits.

```
/* Linker-provided symbols (from LD script) */
extern uint32_t _sstack[];
extern uint32_t _estack[];
extern uint32_t _sheap[];
extern uint32_t _eheap[];

/* Stack overflow check using watermark pattern */
#define STACK_CANARY_VALUE  0xDEADBEEF

void BMS_InitStackCanary(void) {
    uint32_t *p = (uint32_t *)_sstack;
    while(p < (uint32_t *)_estack) {
        *p++ = STACK_CANARY_VALUE;
    }
}

bool BMS_CheckStackOverflow(void) {
    /* Check canary pattern at stack bottom */
    if(*(uint32_t *)_sstack != STACK_CANARY_VALUE) {
        DTC_SetHardFault(DTC_STACK_OVERFLOW_DETECTED);
        return true;
    }
    return false;
}

/* Runtime heap boundary check (wrap pvPortMalloc) */
void* BMS_HeapMalloc(size_t size) {
    void *p = pvPortMalloc(size);
    if(p == NULL) {
        DTC_SetHardFault(DTC_HEAP_EXHAUSTED);
        BMS_EnterSafeState(SAFE_STATE_REASON_MEMORY_FAULT);
    }
    /* Verify allocation stays within heap bounds */
    if(p < (void *)_sheap || p > (void *)_eheap) {
        DTC_SetHardFault(DTC_HEAP_CORRUPTION);
    }
    return p;
}
```

The **Stack Canary** technique fills unused stack memory with a known pattern at startup. A periodic check (every 100 ms via the safety task) verifies the canary value at the stack boundary. If the pattern is corrupted, stack overflow has occurred. The FreeRTOS stack overflow hook `vApplicationStackOverflowHook()` provides an additional layer of protection.

Chapter 4: Looptime Mapping

4.1 Real-Time Task Scheduling

The BMS firmware operates as a multi-rate real-time system where different functions execute at distinct periodicities based on their safety criticality and response requirements. The scheduling strategy uses FreeRTOS software timers and delay-until patterns to maintain precise timing.

The scheduling hierarchy follows a **rate-monotonic** approach: tasks with shorter periods receive higher priorities. This ensures that safety-critical functions (1 ms safety check, 10 ms ADC sampling) always preempt lower-priority background tasks (SOC calculation, logging).

```
/* Rate-monotonic task creation */
void BMS_CreateTasks(void) {
    /* Highest priority: 1 ms safety task */
    xTaskCreate(vTaskSafety, "BMS_Safety", 256, NULL,
                configMAX_PRIORITIES - 1, &hSafety);

    /* High priority: 10 ms ADC + CAN */
    xTaskCreate(vTaskAdcRead, "BMS_AdcRead", 192, NULL,
                configMAX_PRIORITIES - 2, &hAdcRead);
    xTaskCreate(vTaskCanTx, "BMS_CanTx", 128, NULL,
                configMAX_PRIORITIES - 2, &hCanTx);

    /* Medium priority: 100 ms SOC */
    xTaskCreate(vTaskSocCalc, "BMS_SocCalc", 512, NULL,
                configMAX_PRIORITIES - 3, &hSocCalc);

    /* Low priority: 1000 ms background */
    xTaskCreate(vTaskBalance, "BMS_Balance", 256, NULL, 1, &hBalance);
    xTaskCreate(vTaskLogger, "BMS_Logger", 384, NULL, 0, &hLogger);
}
```

4.2 Task Timing Table

Table 4.1: BMS Task Timing Specification

Task Name	Period (ms)	Priority	Stack (words)	WCET (us)	Deadline (ms)
BMS_Safety	1	Highest (4)	256	45	1
BMS_AdcRead	10	High (3)	192	120	10
BMS_SocCalc	100	Medium (2)	512	850	100
BMS_CanTx	10	High (3)	128	35	10

BMS_Balance	1000	Low (1)	256	200	1000
BMS_Logger	1000	Lowest (0)	384	150	1000

The **schedulability test** for rate-monotonic scheduling uses the utilization bound:

```
Total Utilization U = SUM(WCET_i / Period_i)
                        = 0.045/1 + 0.120/10 + 0.850/100 + 0.035/10 + 0.200/1000 +
0.150/1000
                    = 0.045 + 0.012 + 0.0085 + 0.0035 + 0.0002 + 0.00015
                    = 0.06935 (6.94%)

Liu & Layland bound for n=6 tasks: U <= 6*(2^(1/6)-1) = 0.735

U = 0.069 < 0.735 --> SCHEDULABLE (with significant margin)
```

The extremely low CPU utilization (6.94%) is typical for BMS applications where most time is spent waiting for ADC conversions and CAN bus arbitration. The large margin provides headroom for interrupt processing and future feature expansion.

4.3 ADC Sampling Timing

The ADC sampling chain involves sequential conversions of cell voltages, pack current, and temperature sensors. Each conversion requires specific settling times based on the input impedance and sampling capacitor characteristics.

Table 4.2: ADC Channel Timing Breakdown

Signal	Channels	Conversion Time (us)	Settling Time (us)	Total (us)
Cell Voltages (16S)	16	2.5 x 16 = 40	10	50
Pack Current (2x)	2	2.5 x 2 = 5	5	10
Temperature (8x NTC)	8	2.5 x 8 = 20	20	40
HV Rail	1	2.5	5	7.5
Reference	2	2.5 x 2 = 5	0	5
TOTAL (per cycle)	**29**	**72.5**	**40**	**112.5**

At the 10 ms ADC task period, the conversion sequence occupies 112.5 us, representing only 1.1% of the available time window. This allows oversampling (averaging 4 samples per channel) within the 10 ms budget, improving measurement noise performance by 6 dB (factor of 2 in effective resolution).

```
/* ADC timing-controlled sampling with DMA */
void vTaskAdcRead(void *pvParameters) {
    TickType_t xLastWakeTime = xTaskGetTickCount();

    for(;;) {
        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(10));

        uint32_t tStart = BMS_GetMicros();

        /* Trigger cell voltage scan via PDB + ADC + DMA */
        ADC_StartGroupConversion(ADC_CELL_VOLTAGE_GROUP);
        ulTaskNotifyTake(pdTRUE, pdMS_TO_TICKS(5));

        /* Process results - DMA has transferred to buffer */
        BMS_ProcessCellVoltages(adcCellBuffer);

        uint32_t tElapsed = BMS_GetMicros() - tStart;
        if(tElapsed > 5000) { /* 5 ms budget check */
            DTC_SetSoftFault(DTC_ADC_TIMING_VIOLATION);
        }

        /* Timing jitter measurement */
        BMS_RecordTimingJitter(TASK_ADC, tElapsed);
    }
}
```

4.4 CAN Message Periodicity

CAN bus communication follows a periodic transmission schedule with different message types broadcast at distinct rates. The FlexCAN module on the S32K144 supports hardware-based message buffers with automatic retransmission, offloading the CPU from periodic Tx management.

Table 4.3: CAN Message Timing Matrix

Message	CAN ID	DLC	Period (ms)	Priority	Bus Load %
BMS_Status	0x180	8	10	High	1.28
CellVoltages_1_4	0x288	8	100	Medium	0.13
CellVoltages_5_8	0x289	8	100	Medium	0.13
CellVoltages_9_12	0x28A	8	100	Medium	0.13
CellVoltages_13_16	0x28B	8	100	Medium	0.13
Temperatures	0x290	8	100	Medium	0.13
BMS_Diagnostics	0x310	8	100	Low	0.13
BMS_Fault	0x380	8	Event	Critical	<0.01

Total bus load at 500 kbps: approximately 2.1% -- well within the recommended 30% maximum for CAN 2.0B networks. Event-triggered fault messages use a dedicated high-priority mailbox (MB0) to ensure immediate transmission without queueing delay.

4.5 Worst-Case Execution Time Analysis

WCET analysis determines the maximum time a task can execute, accounting for all possible code paths, cache effects, and interrupt interference. For BMS safety certification, WCET must be measured (not just estimated) using hardware tracing or cycle-accurate simulation.

```
/* WCET measurement using DWT cycle counter */
#define DWT_CYCCNT  (*(volatile uint32_t *)0xE0001004)

typedef struct {
    uint32_t wcetUs;           /* Worst case observed */
    uint32_t bcetUs;           /* Best case observed */
    uint32_t avgUs;            /* Rolling average */
    uint32_t jitterUs;         /* Max jitter observed */
    uint32_t violationCount;   /* Deadline violations */
} TimingProfile_t;

TimingProfile_t taskTiming[TASK_COUNT];

void BMS_MeasureWCET(TaskID_t task, uint32_t startCycles) {
    uint32_t elapsed = (DWT_CYCCNT - startCycles) / 80; /* 80 MHz */
    TimingProfile_t *t = &taskTiming[task];

    if(elapsed > t->wcetUs) t->wcetUs = elapsed;
    if(elapsed < t->bcetUs) t->bcetUs = elapsed;

    /* Exponential moving average: alpha = 1/16 */
    t->avgUs += (elapsed - t->avgUs) >> 4;

    /* Jitter: deviation from expected period */
    int32_t jitter = (int32_t)elapsed - (int32_t)t->avgUs;
    if(abs(jitter) > t->jitterUs) t->jitterUs = abs(jitter);
}

/* Safety check: WCET must not exceed deadline margin */
bool BMS_VerifyTimingBudget(void) {
    bool ok = true;
    for(int i = 0; i < TASK_COUNT; i++) {
        uint32_t deadlineUs = taskPeriod[i] * 1000;
        if(taskTiming[i].wcetUs > deadlineUs * 0.8f) {
            DTC_SetSoftFault(DTC_TIMING_BUDGET_EXCEEDED);
            ok = false;
        }
    }
    return ok;
}
```

Table 4.4: Measured WCET vs Budget (10k cycle sample)

Task	Budget (us)	WCET (us)	BCET (us)	Average (us)	Margin %
BMS_Safety	800	45	38	41	94.4
BMS_AdcRead	8000	120	98	108	98.5
BMS_SocCalc	80000	850	720	780	98.9
BMS_CanTx	8000	35	28	31	99.6
BMS_Balance	800000	200	165	180	99.9

BMS_Logger	800000	150	120	135	99.9

Chapter 5: Segger SystemView Setup

5.1 Installation and Configuration

Segger SystemView is a real-time recording and visualization tool for embedded systems running FreeRTOS. It captures task switches, interrupt entries/exits, API calls, and user-defined events without adding significant overhead to the target application. SystemView is the recommended profiling tool for BMS firmware timing analysis.

Prerequisites: Segger J-Link debugger, SystemView host application (v3.50+), and the SystemView target source package integrated into the firmware build. The S32K144 debug interface uses SWD (Serial Wire Debug) via the J-Link probe.

Table 5.1: SystemView Component Files

File	Location	Purpose
SEGGER_SYSVIEW.c	Segger/SystemView/	Core logging engine
SEGGER_RTT.c	Segger/RTT/	Real-time transfer buffer
SEGGER_SYSVIEW_FreeRTOS.c	Segger/SystemView/	FreeRTOS integration hooks
SEGGER_SYSVIEW_Config_FreeRTOS.c	Config/	Application-specific config

5.2 RTT Buffer Setup

Real-Time Transfer (RTT) uses a ring buffer in target RAM that the J-Link probe reads via debug interface without halting the CPU. RTT requires dedicated RAM for the up-buffer (target to host) and down-buffer (host to target). The recommended buffer size for BMS profiling is 4 KB up-buffer and 512 B down-buffer.

```
/* SystemView configuration for S32K144 */
#define SYSVIEW_RTT_CHANNEL        1
#define SYSVIEW_BUFFER_SIZE        4096
#define SYSVIEW_TIMESTAMP_FREQ     80000000  /* 80 MHz core clock */
#define SYSVIEW_CPU_FREQ           80000000

/* RTT buffer allocation in RAM */
static uint8_t _acUpBuffer[SYSVIEW_BUFFER_SIZE];
static uint8_t _acDownBuffer[512];

/* Application ID and description */
#define SYSVIEW_APP_NAME    "BMS S32K144 v2.3"
#define SYSVIEW_DEVICE_NAME    "NXP S32K144"

void SEGGER_SYSVIEW_Conf(void) {
    SEGGER_RTT_CONFIG_UP_BUFFER(SYSVIEW_RTT_CHANNEL,
                                _acUpBuffer,
                                sizeof(_acUpBuffer),
                                SEGGER_RTT_MODE_NO_BLOCK_SKIP);

    SEGGER_RTT_CONFIG_DOWN_BUFFER(SYSVIEW_RTT_CHANNEL,
                                  _acDownBuffer,
                                  sizeof(_acDownBuffer),
                                  SEGGER_RTT_MODE_NO_BLOCK_SKIP);

    SEGGER_SYSVIEW_Init(SYSVIEW_TIMESTAMP_FREQ,
                        SYSVIEW_CPU_FREQ,
                        &_SYSVIEW_X_FreeRTOS,
                        _cbSendSystemDesc);

    SEGGER_SYSVIEW_SetRAMBase(0x1FFF8000);
    SEGGER_SYSVIEW_Start();
}
```

5.3 Event Logging

SystemView captures three categories of events automatically: OS events (task switches, semaphore operations, queue access), interrupt events (entry/exit with ID), and API trace (FreeRTOS API calls). User events can be added to mark specific BMS operations.

```
/* User event IDs for BMS-specific logging */
#define SYSVIEW_EVT_ADC_START       100
#define SYSVIEW_EVT_ADC_COMPLETE    101
#define SYSVIEW_EVT_CAN_TX          102
#define SYSVIEW_EVT_CAN_RX          103
#define SYSVIEW_EVT_BALANCE_START   104
#define SYSVIEW_EVT_BALANCE_DONE    105
#define SYSVIEW_EVT_FAULT_SET       106
#define SYSVIEW_EVT_CONTACTOR_CMD   107

/* Recording macros (zero-cost when disabled) */
#define BMS_TRACE_EVENT(id)      SEGGER_SYSVIEW_RecordVoidEvent(id)
#define BMS_TRACE_VALUE(id,val)  SEGGER_SYSVIEW_RecordU32(id, val)

/* Usage example in ADC task */
void vTaskAdcRead(void *pvParameters) {
    for(;;) {
        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(10));

        BMS_TRACE_EVENT(SYSVIEW_EVT_ADC_START);
        ADC_StartGroupConversion(ADC_CELL_VOLTAGE_GROUP);

        ulTaskNotifyTake(pdTRUE, pdMS_TO_TICKS(5));
        BMS_TRACE_EVENT(SYSVIEW_EVT_ADC_COMPLETE);

        uint16_t maxCell = BMS_GetMaxCellVoltage();
        BMS_TRACE_VALUE(SYSVIEW_EVT_ADC_COMPLETE, maxCell);

        BMS_ProcessCellVoltages(adcBuffer);
    }
}
```

5.4 Timeline Analysis

The SystemView timeline view displays all recorded events on a horizontal time axis, enabling visual identification of timing anomalies, priority inversions, and excessive interrupt latency. For BMS analysis, focus on these key metrics:

Table 5.2: SystemView Timing Metrics

Metric	Target	Analysis Method
Task period jitter	< 5% of period	Measure delta between successive starts
Context switch time	< 20 us	Time between task exit and next entry
Interrupt latency	< 2 us	Time from signal to ISR entry
ISR duration	< 50 us	Total time in interrupt handler

CPU load	< 15%	Idle task run-time percentage
Priority inversion	None	Verify no lower-priority task blocks higher

5.5 Profiling a BMS Session

Follow this procedure to capture and analyze a representative BMS firmware session:

1. **Build with profiling enabled**: Set `configUSE_TRACE_FACILITY = 1` and `configGENERATE_RUN_TIME_STATS = 1` in FreeRTOSConfig.h
2. **Initialize SystemView**: Call `SEGGER_SYSVIEW_Conf()` after scheduler start
3. **Start recording**: Launch SystemView host application, connect J-Link, press Record
4. **Exercise all code paths**: Trigger ADC conversions, send CAN messages, inject fault conditions
5. **Stop and save**: Save the recording (.SVDat) for offline analysis and regression testing

> **Performance Impact**
>
> SystemView adds approximately 1-2 us per logged event and consumes 4 KB of RAM for the RTT buffer. At 1000 events per second, the CPU overhead is less than 0.2%. Disable SystemView in production builds by setting `configUSE_SEGGER_SYSTEMVIEW = 0`.

Chapter 6: MIL vs PIL Testing Architecture

6.1 Model-in-the-Loop Overview

Model-in-the-Loop (MIL) testing validates the BMS control algorithm in a pure simulation environment before any hardware or embedded code exists. The BMS model, developed in MATLAB/Simulink, executes on the host PC with the plant model (battery pack, thermal dynamics, aging behavior) running in the same simulation loop.

The MIL environment enables thorough validation of SOC estimation algorithms, cell balancing strategies, and thermal management logic against a battery model that can simulate edge cases impractical to test on physical hardware: internal short circuits, extreme temperature gradients, and end-of-life cell behavior.

```
/* MIL test harness structure */
Simulink Model:
  +-- BMS_Controller.slx (production algorithm)
  +-- Battery_Plant.slx  (equivalent circuit model)
  +-- Test_Sequences.m   (drive cycles, fault injection)
  +-- PassFail_Criteria.m (tolerance definitions)

Test Coverage:
  - UDDS drive cycle (1372 seconds, urban traffic)
  - US06 drive cycle (596 seconds, aggressive driving)
  - Static capacity test (0.1C discharge, 25 deg C)
  - Cold start (-20 deg C, 1C discharge)
  - Hot soak (50 deg C, 2C charge)
  - Cell imbalance (100 mV initial delta)
```

Table 6.1: MIL Test Coverage Requirements

Test Category	Scenarios	Pass Criteria	Priority
SOC Estimation	15	Max error < 3% at EOL	Critical
Cell Balancing	8	Delta < 20 mV at end	Critical
Thermal Model	10	T error < 2 deg C	High
Fault Detection	20	100% detection rate	Critical
Contactor Logic	12	No weld events	Critical
Current Limit	6	No overshoot > 5%	High

6.2 Processor-in-the-Loop

Processor-in-the-Loop (PIL) testing executes compiled production code on the actual target microcontroller (S32K144) while the plant model continues to run on the host PC. The host and target communicate via serial (UART) or debug interface (J-Link RTT), exchanging sensor inputs and actuator outputs each simulation step.

PIL serves as the critical bridge between simulation and hardware-in-the-loop (HIL) testing. It validates that:

- Code generation produces functionally correct embedded C
- Fixed-point arithmetic does not degrade algorithm accuracy
- Target compiler optimizations do not alter behavior
- Execution timing fits within the real-time budget

```
/* PIL communication protocol */
Host (Simulink) <--> Target (S32K144)

Message Frame (16 bytes):
  [0:3]   uint32_t sequenceNumber
  [4:7]   float32_t cellVoltages[16]  -- simulated ADC counts
  [8:11]  float32_t packCurrent       -- simulated shunt value
  [12:15] float32_t temperatures[8]   -- simulated NTC values
  [16:19] float32_t bmsCommand        -- contactor/balance output

Timing: Host sends inputs, waits for target response.
        Timeout = 2x expected WCET triggers test failure.
```

6.3 Test Vector Generation

Test vectors are generated from MIL simulation recordings and replayed during PIL and HIL testing. Each vector contains the complete state of all BMS inputs at a given timestep, along with the expected outputs computed by the golden model.

```
/* Test vector file format (.csv) */
Time_ms, V_cell1_mV, V_cell2_mV, ..., I_pack_mA, T_max_C,
        Expected_SOC, Expected_balance, Expected_fault

0,      4200, 4195, ...,     0, 25,  100.0, 0, 0
100,    4198, 4192, ..., -5000, 25,   99.8, 0, 0
200,    4195, 4188, ..., -5000, 25,   99.6, 1, 0
...

/* Automated test runner */
bool BMS_RunTestVector(const char *vectorFile) {
    FILE *f = fopen(vectorFile, "r");
    TestRecord_t rec;
    bool pass = true;

    while(ReadTestRecord(f, &rec)) {
        /* Apply inputs to BMS */
        BMS_SetCellVoltages(rec.cellVoltages);
        BMS_SetCurrent(rec.packCurrent);
        BMS_SetTemperatures(rec.temperatures);

        /* Execute one BMS cycle */
        BMS_RunCycle(rec.timeDelta_ms);

        /* Verify outputs against expected */
        if(abs(BMS_GetSOC() - rec.expectedSOC) > 0.5f) {
            LogFail("SOC mismatch at t=%u", rec.time_ms);
            pass = false;
        }
        if(BMS_GetBalanceState() != rec.expectedBalance) {
            LogFail("Balance mismatch at t=%u", rec.time_ms);
            pass = false;
        }
    }
    fclose(f);
    return pass;
}
```

6.4 Automated Test Reporting and CI/CD Integration

Integrate PIL testing into the CI/CD pipeline using Jenkins, GitLab CI, or GitHub Actions. Each firmware commit triggers a complete test suite: static analysis (MISRA-C check), unit tests (Unity framework), PIL regression tests, and code coverage reporting.

```
# GitLab CI pipeline for BMS firmware
stages:
  - build
  - static_analysis
  - unit_test
  - pil_test
  - report

pil_test_job:
  stage: pil_test
  tags: [s32k144-hil-bench]
  script:
    - make pil TARGET=S32K144
    - python3 scripts/run_pil_suite.py
          --vectors test_vectors/gold/
          --device /dev/ttyACM0
          --output results/pil_report.xml
    - python3 scripts/check_coverage.py
          --input results/pil_report.xml
          --threshold 95
  artifacts:
    reports:
      junit: results/pil_report.xml
    paths:
      - results/pil_coverage.html
```

Table 6.2: CI/CD Test Stage Summary

Stage	Duration	Threshold	Fail Action
Build	2 min	Zero warnings	Reject commit
MISRA-C Check	3 min	Zero violations	Reject commit
Unit Tests	5 min	100% pass	Reject commit
PIL Regression	15 min	95% coverage	Flag for review
Code Coverage	1 min	Branch > 90%	Flag for review

Chapter 7: Diagnostic Trouble Codes (DTC)

7.1 UDS DTC Format

Diagnostic Trouble Codes in automotive applications follow the ISO 15031-6 standard, encoded as 3-byte values (DTC High, DTC Middle, DTC Low). The DTC structure encodes both the fault category and the specific failing system.

```
DTC Byte Structure (24 bits):

DTC High Byte:
  Bit 7-6: Category (00=P, 01=C, 10=B, 11=U)
  Bit 5-4: First digit (0-3)
  Bit 3-0: Second digit (0-F)

DTC Middle Byte:
  Bit 7-0: Third and fourth hex digits

DTC Low Byte:
  Bit 7-6: Failure Type (00=general, 01=circuit, ...)
  Bit 5-0: Fault qualifier

Categories:
  P = Powertrain (battery, motor, inverter)
  C = Chassis
  B = Body
  U = Network communication
```

For BMS applications, the majority of DTCs use the **P0Axx** range reserved for hybrid and electric vehicle battery systems. The complete DTC is packed into a 3-byte array for UDS Service $19 (ReadDTCInformation) responses.

7.2 BMS-Specific DTC Definitions

Table 7.1: BMS Diagnostic Trouble Codes

DTC	Name	Detection Condition	MIL	Safe State
P0A01	Cell Voltage Imbalance	Max cell delta > 100 mV for 5 s	On	Disable charge
P0A02	Temperature Sensor Fault	NTC open/short or out of range (-40 to +85 deg C valid)	On	Limit current
P0A03	Overcurrent Condition	|I_pack| > I_limit for 100 ms	On	Open contactors

P0A04	Isolation Fault	Isolation resistance < 500 Ohm/V	On	Open contactors
P0A05	Contactor Weld Detected	V_batt present when contactor commanded open	On	Disable all
P0A0F	Flash ECC Single-Bit	SECDED correctable error detected	Off	Log only
P0A10	Flash ECC Double-Bit	SECDED uncorrectable error detected	On	Open contactors
P0A11	Stack Overflow	Canary pattern corrupted or FreeRTOS hook triggered	On	Reset
P0A12	Heap Exhaustion	pvPortMalloc() returns NULL	On	Limited function
P0A13	ADC Timing Violation	Conversion sequence exceeds 5 ms budget	Off	Log only
P0A14	CAN Bus Off	TEC > 255 (bus off condition)	On	Derate power
P0A15	SOC Estimation Divergence	|SOC_coulomb - SOC_EKF| > 5%	Off	Recalibrate
P0A16	Timing Budget Exceeded	WCET > 80% of task period	Off	Log only

7.3 DTC Storage Strategy

DTC records must persist across power cycles and be retrievable via the diagnostic interface. The storage strategy uses a circular buffer in the flash configuration region (see Chapter 2), with each record containing the full diagnostic snapshot.

```
/* DTC record structure (32 bytes) */
typedef struct {
    uint16_t dtcCode;           /* 2 bytes: DTC High + Middle */
    uint8_t  dtcLow;            /* 1 byte: DTC Low byte */
     uint8_t  statusByte;         /* 1 byte: UDS status (testFailed, confirmed, ...)
*/
    uint32_t timestamp;         /* 4 bytes: RTC epoch seconds */
    uint32_t occurrenceCount;   /* 4 bytes: total occurrences */
       int16_t   snapshotCellV[16];  /*  32  bytes:  cell  voltages  at  fault  (stored
separately) */
    int16_t  snapshotCurrent;   /* 2 bytes: pack current */
    int8_t   snapshotTempMax;   /* 1 byte: max temperature */
    uint8_t  snapshotSOC;       /* 1 byte: SOC percentage */
} DTC_Record_t;

/* Status byte bit definitions (ISO 14229-1) */
#define DTC_STATUS_TEST_FAILED              0x01
#define DTC_STATUS_TEST_FAILED_THIS_CYCLE   0x02
#define DTC_STATUS_PENDING                  0x04
#define DTC_STATUS_CONFIRMED                0x08
#define DTC_STATUS_TEST_NOT_COMPLETED       0x10
#define DTC_STATUS_TEST_FAILED_SINCE_CLEAR  0x20
#define DTC_STATUS_TEST_NOT_COMPLETED_SINCE_CLEAR 0x40
#define DTC_STATUS_WARNING_INDICATOR        0x80

/* Store DTC with snapshot data */
void DTC_Store(uint16_t code, uint8_t low, const DTC_Snapshot_t *snapshot) {
    DTC_Record_t rec;
    rec.dtcCode = code;
    rec.dtcLow = low;
    rec.statusByte = DTC_STATUS_CONFIRMED | DTC_STATUS_TEST_FAILED;
    rec.timestamp = RTC_GetEpochSeconds();
    rec.snapshotCurrent = snapshot->packCurrent;
    rec.snapshotTempMax = snapshot->maxTemperature;
    rec.snapshotSOC = snapshot->socPercent;

    /* Check for existing record to increment counter */
    DTC_Record_t *existing = DTC_Find(code, low);
    if(existing) {
        rec.occurrenceCount = existing->occurrenceCount + 1;
    } else {
        rec.occurrenceCount = 1;
    }

    Flash_WriteRecord(FLASH_REGION_DTC, &rec, sizeof(rec));
}
```

Storage Capacity

A 64 KB configuration region with 32-byte DTC records supports approximately 2000 stored DTC entries. At a typical fault rate of 10 DTCs per day during development, this provides 200 days of storage. For production, periodic clearing via Service $14 (ClearDiagnosticInformation) is required.

Chapter 8: Production Test Procedures

8.1 ICT Flash Programming

In-Circuit Test (ICT) flash programming loads the initial firmware and calibration data into the S32K144 during PCB assembly. The programming fixture uses the JTAG/SWD interface or the built-in UART bootloader to write the flash image before any functional testing begins.

The flash programming sequence follows this order:

1. Enter bootloader mode via TEST pin assertion or BDM command
2. Erase entire program flash (mass erase command)
3. Program bootloader image (32 KB) at base address 0x0000_0000
4. Program application image (384 KB) at address 0x0000_8000
5. Verify both regions using CRC-32 checksum
6. Write initial configuration block with default calibration values
7. Lock flash security to prevent unauthorized readout
8. Release reset and verify application startup via UART heartbeat

```
/* ICT flash programming parameters */
#define ICT_PROGRAMMING_TIMEOUT_MS    30000
#define ICT_VERIFY_CHUNK_SIZE         4096
#define ICT_MAX_RETRIES               3

bool ICT_ProgramFlash(const uint8_t *image, uint32_t size, uint32_t addr) {
    Boot_EnterMode(BOOT_MODE_JTAG);

    if(!Flash_MassErase()) return false;

    uint32_t offset = 0;
    while(offset < size) {
        uint32_t chunk = (size - offset > ICT_VERIFY_CHUNK_SIZE)
                       ? ICT_VERIFY_CHUNK_SIZE : (size - offset);

        for(int retry = 0; retry < ICT_MAX_RETRIES; retry++) {
            if(Flash_Write(addr + offset, image + offset, chunk)) {
                if(Flash_Verify(addr + offset, image + offset, chunk)) {
                    break; /* Success */
                }
            }
            if(retry == ICT_MAX_RETRIES - 1) return false;
        }
        offset += chunk;
    }

    /* Final CRC verification over entire image */
    uint32_t crcCalc = CRC32_Calculate(image, size);
    uint32_t crcExpected = *(uint32_t *)(image + size - 4);
    return (crcCalc == crcExpected);
}
```

8.2 EOL Functional Test Sequence

End-of-Line (EOL) testing validates the complete BMS functionality after PCB assembly, flash programming, and mechanical integration. The EOL sequence exercises all analog inputs, digital outputs, communication interfaces, and safety functions.

Table 8.1: EOL Functional Test Sequence

Step	Test	Method	Pass Criteria	Timeout (ms)
1	Power-on self test	Internal	All tests pass, DTC count = 0	500
2	Cell voltage ADC	Apply 3.0-4.2 V / cell	Error < 5 mV per cell	200
3	Current sensor	Apply +/- 100 A equivalent	Error < 0.5% FS	200
4	Temperature sensor	Apply 25 deg C equivalent	Error < 1 deg C	200

5	HV isolation	Apply 500 V, measure leakage	> 500 Ohm/V	1000
6	Contactor drive	Command on/off, sense feedback	Correct state, no weld	500
7	CAN communication	Send/receive test frames	0 errors, latency < 5 ms	200
8	Precharge	Command sequence, verify timing	V_cap reaches 95% in < 500 ms	1000
9	Cell balancing	Enable all channels, measure	Current > 50 mA per channel	500
10	Fault injection	Simulate overvoltage, UV, OT	All faults detected within 100 ms	500
11	Flash write/verify	Write test pattern, read back	100% match, ECC clean	500
12	RTC / timestamp	Set and read back time	Error < 1 s after 60 s	60000

8.3 HiPot and Insulation Testing

HiPot (High Potential) testing applies a high voltage between the HV+ bus and chassis ground to verify insulation integrity. The BMS must survive this test without damage and report correct isolation resistance afterward.

Test parameters for a 400 V nominal battery pack:

- **Test voltage**: 2 x V_max + 1000 V = 1800 V DC
- **Ramp rate**: 500 V/s maximum
- **Dwell time**: 60 seconds at full voltage
- **Leakage limit**: < 1 mA
- **Post-test isolation**: > 500 Ohm/V (per ECE R100)

During HiPot testing, the BMS isolation monitoring circuit (typically a resistor bridge with MCU ADC measurement) must be disconnected or protected to prevent damage. After the HiPot test completes, the BMS runs a self-test to verify isolation measurement accuracy.

8.4 Calibration Procedures

Production calibration compensates for component tolerances in the analog measurement chain. Each BMS unit requires individual calibration for cell voltage gain, current sensor offset/gain, and temperature sensor curves.

```
/* Calibration parameter storage */
typedef struct {
    float cellVoltageGain[16];      /* Per-cell gain correction */
    float cellVoltageOffset[16];    /* Per-cell offset in mV */
    float currentOffset;            /* Zero-current offset */
    float currentGain;              /* A-to-ADC counts gain */
    float tempBeta[8];              /* NTC beta values */
    float tempOffset[8];            /* Temperature offsets */
    uint32_t calTimestamp;          /* When calibration performed */
    uint16_t calStationId;          /* EOL station identifier */
    uint16_t crc16;                 /* Parameter integrity check */
} CalibrationData_t;

/* Current sensor calibration procedure */
void CAL_CurrentSensor(BMS_Channel_t ch) {
    /* Step 1: Zero-current offset */
    int32_t sum = 0;
    for(int i = 0; i < 1000; i++) {
        sum += ADC_ReadCurrent(ch);
        Delay_ms(1);
    }
    calData.currentOffset = sum / 1000.0f;

    /* Step 2: Positive gain (apply +50 A reference) */
    float adcPlus = ADC_ReadCurrent(ch);
    /* Step 3: Negative gain (apply -50 A reference) */
    float adcMinus = ADC_ReadCurrent(ch);

    float span = (adcPlus - adcMinus);
    calData.currentGain = 100.0f / span; /* A per ADC count */

    /* Store to flash */
    Flash_WriteCalibration(&calData);
}
```

Table 8.2: Calibration Accuracy Requirements

Parameter	Pre-Cal Accuracy	Post-Cal Accuracy	Reference
Cell voltage	+/- 10 mV	+/- 1 mV	6.5-digit DMM
Pack current	+/- 2% FS	+/- 0.3% FS	Calibrated shunt
Temperature	+/- 3 deg C	+/- 0.5 deg C	RTD probe
Isolation	+/- 10%	+/- 5%	Standard resistor

Chapter 9: Field Diagnostics

9.1 OBD-II CAN Diagnostic Session

Field diagnostics of the BMS use the standardized OBD-II communication interface over CAN bus at 500 kbps (CAN ID 0x7E0 for tester requests, 0x7E8 for BMS responses). The diagnostic session follows ISO 15765-4 (CAN-based transport) and ISO 14229 (UDS services).

To initiate a diagnostic session, the tester sends UDS Service $10 (DiagnosticSessionControl) with the session type parameter:

```
/* Diagnostic session types */
#define DEFAULT_SESSION         0x01
#define PROGRAMMING_SESSION     0x02
#define EXTENDED_SESSION        0x03
#define SAFETY_SYSTEM_SESSION   0x04

/* Example: Request extended diagnostic session */
Tester -> BMS:  7E0  02 10 03 00 00 00 00 00
BMS -> Tester: 7E8  06 50 03 00 14 01 F4 00

Response decoded:
  0x50 = positive response to Service $10
  0x03 = extended session confirmed
  0x0014 = P2 timeout (20 ms)
  0x01F4 = P2* timeout (5000 ms, enhanced)
```

9.2 Live Parameter Reading

Service $22 (ReadDataByIdentifier) retrieves live BMS parameters using 2-byte Data Identifiers (DIDs). The S32K144 firmware maintains a DID table mapping identifiers to RAM-resident variables.

Table 9.1: BMS Data Identifier Mapping

DID	Name	Size	Data Type	Update Rate
F100	Pack Voltage	2	uint16_t (0.1 V)	10 ms
F101	Pack Current	2	int16_t (0.1 A)	10 ms
F102	Max Cell Voltage	2	uint16_t (1 mV)	100 ms
F103	Min Cell Voltage	2	uint16_t (1 mV)	100 ms
F104	Avg Cell Temperature	1	int8_t (deg C)	100 ms
F105	Max Cell Temperature	1	int8_t (deg C)	100 ms

F106	SOC	1	uint8_t (0.5%)	100 ms
F107	SOH	1	uint8_t (0.5%)	1000 ms
F108	Isolation Resistance	2	uint16_t (kOhm)	1000 ms
F109	Contactor State	1	uint8_t (bitfield)	10 ms
F10A	Balance Status	2	uint16_t (bitmask)	1000 ms
F10B	DC Fault Count	2	uint16_t	Event

```
/* DID handler implementation */
void UDS_HandleReadDataByIdentifier(uint16_t did, uint8_t *out, uint8_t *len) {
    switch(did) {
        case 0xF100:
            *(uint16_t *)out = (uint16_t)(BMS_GetPackVoltage() * 10);
            *len = 2;
            break;
        case 0xF101:
            *(int16_t *)out = (int16_t)(BMS_GetPackCurrent() * 10);
            *len = 2;
            break;
        case 0xF106:
            out[0] = (uint8_t)(BMS_GetSOC() * 2);
            *len = 1;
            break;
        /* ... additional DIDs ... */
        default:
            UDS_SendNegativeResponse(SID_READ_DATA,
                                     NRC_REQUEST_OUT_OF_RANGE);
            *len = 0;
    }
}
```

9.3 Fault Retrieval

Service $19 (ReadDTCInformation) retrieves stored DTCs with their status, timestamps, and snapshot data. The BMS supports subfunctions $02 (report DTC by status mask) and $04 (report DTC snapshot record).

```
/* UDS DTC retrieval sequence */
Tester  -> BMS:  7E0  03 19 02 FF 00 00 00 00
BMS  -> Tester: 7E8  10 12 59 02 FF 0A 01 00
BMS  -> Tester: 7E8  21 0A 02 00 66 54 32 10

Response decoded (single frame):
  0x59 = positive response to Service $19
  0x02 = report by status mask
  0xFF = status mask (all DTCs)
  DTC #1: 0x0A01 00 -- P0A01, status: test failed
           timestamp: 0x66543210 (epoch)
  DTC #2: 0x0A02 00 -- P0A02, status: test failed
           timestamp: ...
```

9.4 Data Transfer for Log Download

UDS Services $34 (RequestDownload), $35 (RequestUpload), and $36 (TransferData) enable downloading the flash-based event log from the BMS to the diagnostic tester. The log contains time-stamped records of cell voltages, temperatures, current, and fault events.

```
/* Log download sequence */
Tester -> BMS:  34 00 44 00 07 80 00 00  /* Request upload, addr=0x78000, len=32KB
*/
BMS -> Tester: 74 10                     /* Positive, max block = 256 bytes */

Tester -> BMS:  36 01                    /* Transfer data, block seq = 1 */
BMS -> Tester: 76 01 [256 bytes data]
Tester -> BMS:  36 02                    /* Block seq = 2 */
BMS -> Tester: 76 02 [256 bytes data]
...
Tester -> BMS:  37                       /* Request transfer exit */
BMS -> Tester: 77                       /* Transfer complete */
```

The flash logging ring buffer implementation stores records with a fixed-size header for efficient retrieval:

```
/* Flash log ring buffer entry */
typedef struct {
    uint32_t timestamp;       /* RTC epoch seconds */
    uint16_t seqNumber;       /* Monotonic sequence */
    uint8_t  entryType;       /* VOLTAGE, TEMP, CURRENT, FAULT */
    uint8_t  dataLength;      /* Payload bytes following */
    /* Variable-length payload follows */
} LogEntryHeader_t;

/* Ring buffer state in RAM */
typedef struct {
    uint32_t writePointer;    /* Next write address in flash */
    uint32_t readPointer;     /* Last read address */
    uint32_t oldestPointer;   /* Oldest valid entry */
    uint16_t entryCount;      /* Valid entries in buffer */
    bool     wrapOccurred;    /* Buffer has wrapped at least once */
} LogRingState_t;
```

9.5 Remote Diagnostics via Telematics Gateway

Connected vehicles enable remote BMS diagnostics through the telematics control unit (TCU). The TCU acts as a CAN-to-cellular gateway, forwarding diagnostic requests from a remote server to the BMS and returning responses.

Table 9.2: Remote Diagnostic Use Cases

Use Case	Trigger	Data Retrieved	Frequency
Health monitoring	Scheduled	SOC, SOH, DTC count	Daily
Fault alerting	DTC confirmed	Full DTC + snapshot	Immediate
Predictive maint	SOH threshold	Capacity fade trend	Weekly
Recall validation	OEM request	Software version, cal	On demand
Warranty analysis	Claim filed	Full log download	On demand

Chapter 10: Reference Tables

10.1 S32K144 Pin Mapping for BMS Applications

Table 10.1: S32K144 Pin Assignment - BMS Peripheral Mapping

Pin	Port/Pin	Function	BMS Signal	Direction
1	PTA0	ADC0_SE0	Cell 1 Voltage	Analog In
2	PTA1	ADC0_SE1	Cell 2 Voltage	Analog In
3	PTA2	ADC0_SE2	Cell 3 Voltage	Analog In
4	PTA3	ADC0_SE3	Cell 4 Voltage	Analog In
5	PTA6	ADC0_SE6	Cell 5 Voltage	Analog In
6	PTA7	ADC0_SE7	Cell 6 Voltage	Analog In
7	PTA8	ADC0_SE8	Cell 7 Voltage	Analog In
8	PTA9	ADC0_SE9	Cell 8 Voltage	Analog In
9	PTB0	ADC1_SE0	Cell 9 Voltage	Analog In
10	PTB1	ADC1_SE1	Cell 10 Voltage	Analog In
11	PTB2	ADC1_SE2	Cell 11 Voltage	Analog In
12	PTB3	ADC1_SE3	Cell 12 Voltage	Analog In
13	PTB4	ADC1_SE4	Cell 13 Voltage	Analog In
14	PTB5	ADC1_SE5	Cell 14 Voltage	Analog In
15	PTB6	ADC1_SE6	Cell 15 Voltage	Analog In
16	PTB7	ADC1_SE7	Cell 16 Voltage	Analog In
17	PTB10	ADC1_SE14	Pack Current (+)	Analog In
18	PTB11	ADC1_SE15	Pack Current (-)	Analog In
19	PTC0	ADC0_SE14	Temp 1 (NTC)	Analog In
20	PTC1	ADC0_SE15	Temp 2 (NTC)	Analog In
21	PTC14	CAN0_RX	CAN Bus Rx	Digital In
22	PTC15	CAN0_TX	CAN Bus Tx	Digital Out
23	PTC16	CAN1_RX	CAN Diag Rx	Digital In
24	PTC17	CAN1_TX	CAN Diag Tx	Digital Out

25	PTD0	GPIO	Contactor+ Drive	Digital Out
26	PTD1	GPIO	Contactor- Drive	Digital Out
27	PTD2	GPIO	Precharge Drive	Digital Out
28	PTD3	GPIO	Contactor+ Sense	Digital In
29	PTD4	GPIO	Contactor- Sense	Digital In
30	PTD5	FTM0_CH5	Balance PWM Ch1	PWM Out
31	PTD6	FTM0_CH6	Balance PWM Ch2	PWM Out
32	PTD7	FTM0_CH7	Balance PWM Ch3	PWM Out
33	PTE0	LPUART1_TX	Debug UART Tx	Digital Out
34	PTE1	LPUART1_RX	Debug UART Rx	Digital In

10.2 CAN Message Matrix

Table 10.2: BMS CAN Message Matrix (500 kbps)

Message	ID (hex)	DLC	Period	Direction	Priority
BMS_Status1	0x180	8	10 ms	Tx	High
BMS_Status2	0x181	8	10 ms	Tx	High
CellVoltages_1_4	0x288	8	100 ms	Tx	Medium
CellVoltages_5_8	0x289	8	100 ms	Tx	Medium
CellVoltages_9_12	0x28A	8	100 ms	Tx	Medium
CellVoltages_13_16	0x28B	8	100 ms	Tx	Medium
Temperature_1_4	0x290	8	100 ms	Tx	Medium
Temperature_5_8	0x291	8	100 ms	Tx	Medium
BMS_Diagnostics	0x310	8	100 ms	Tx	Low
BMS_Fault	0x380	8	Event	Tx	Critical
VCU_Command	0x100	8	10 ms	Rx	High
VCU_Limits	0x101	8	100 ms	Rx	Medium
Charger_Status	0x200	8	100 ms	Rx	Medium
Diag_Request	0x7E0	8	Event	Rx	High
Diag_Response	0x7E8	8	Event	Tx	High

10.3 FreeRTOS Config Parameter Quick Reference

Table 10.3: FreeRTOSConfig.h - BMS Recommended Values

Parameter	Value	Description	Rationale
configCPU_CLOCK_HZ	80000000	Core clock frequency	S32K144 max
configTICK_RATE_HZ	1000	RTOS tick frequency	1 ms resolution
configMAX_PRIORITIES	8	Priority levels	5 tasks + margins
configMINIMAL_STACK_SIZE	128	Idle task stack (words)	Minimum safe
configTOTAL_HEAP_SIZE	24576	Total heap (bytes)	24 KB heap
configMAX_TASK_NAME_LEN	16	Task name buffer	Debugging
configUSE_TRACE_FACILITY	1	Enable vTaskList()	Profiling
configUSE_STATS_FORMATTING	1	Enable run-time stats	CPU load monitoring
configGENERATE_RUN_TIME_STATS	1	Runtime counter	WCET analysis
configUSE_MALLOC_FAILED_HOOK	1	Trap malloc failures	Heap protection
configCHECK_FOR_STACK_OVERFLOW	2	Comprehensive check	Stack protection
configUSE_QUEUE_SETS	0	Queue set feature	Not needed
configUSE_TASK_NOTIFICATIONS	1	Lightweight sync	ISR-to-task signaling
configUSE_MUTEXES	1	Mutual exclusion	Shared resource protection
configUSE_RECURSIVE_MUTEXES	0	Recursive locks	Not needed
configUSE_COUNTING_SEMAPHORES	1	Counting semaphores	DMA completion
configUSE_TICKLESS_IDLE	0	Low power mode	Always-active BMS
configUSE_APPLICATION_TASK_TAG	0	Task tags	Not needed
configUSE_CO_ROUTINES	0	Co-routine support	Deprecated

10.4 MISRA-C Compliance Checklist

Table 10.4: MISRA-C:2012 Compliance Summary

Rule Category	Required	Advisory	Deviation Required
Standard C environment	Rule 1.1 - 1.3	--	None

Unused code	Rule 2.1 - 2.2	--	ISR vectors (Rule 2.1)
Comments	Rule 3.1 - 3.2	--	None
Character sets	Rule 4.1 - 4.2	--	None
Identifiers	Rule 5.1 - 5.9	--	Register names (Rule 5.1)
Types	Rule 6.1 - 6.2	--	HW register access (Rule 6.2)
Constants	Rule 7.1 - 7.4	--	String literals (Rule 7.4)
Declarations and definitions	Rule 8.1 - 8.14	--	External variables (Rule 8.7)
Initialization	Rule 9.1 - 9.5	--	Union initialization (Rule 9.3)
Operators	Rule 10.1 - 10.8	--	Bitwise on signed (Rule 10.1)
Control flow	Rule 11.1 - 11.6	--	Function pointers (Rule 11.1)
Expressions	Rule 12.1 - 12.4	--	None
Side effects	Rule 13.1 - 13.6	--	Loop counters (Rule 13.3)
Control statement expressions	Rule 14.1 - 14.4	--	None
Control flow	Rule 15.1 - 15.7	--	Switch fallthrough (Rule 15.5)
Switch statements	Rule 16.1 - 16.7	--	Default case (Rule 16.4)
Functions	Rule 17.1 - 17.8	--	Variable arguments (Rule 17.1)
Pointers and arrays	Rule 18.1 - 18.9	--	Register pointers (Rule 18.3)
Overlapping storage	Rule 19.1 - 19.2	--	None
Preprocessor directives	Rule 20.1 - 20.14	--	Include order (Rule 20.1)
Standard libraries	Rule 21.1 - 21.21	--	setjmp (Rule 21.4)
Resources and libraries	Rule 22.1 - 22.10	--	None

10.5 Glossary

ADC

Analog-to-Digital Converter. Converts analog cell voltages and temperatures to digital values for processing.

BMS

Battery Management System. Electronic system that manages a rechargeable battery pack by monitoring its state, protecting it from operating outside safe limits, and balancing cells.

CAN

Controller Area Network. Robust vehicle bus standard for communication between ECUs without a host computer.

DTC

Diagnostic Trouble Code. Standardized fault identifier following ISO 15031-6 format (e.g., P0A01).

ECC

Error Correction Code. SECDED hardware that detects and corrects single-bit flash errors.

ECU

Electronic Control Unit. Embedded computer that controls one or more electrical systems in a vehicle.

EKF

Extended Kalman Filter. Recursive algorithm for SOC estimation that handles non-linear battery dynamics.

EOL

End-of-Line. Final testing performed on a product after manufacturing and before shipment.

Flash

Non-volatile memory that stores firmware and configuration data, electrically erasable and programmable.

FreeRTOS

Open-source real-time operating system kernel for embedded devices, distributed under MIT license.

HIL

Hardware-in-the-Loop. Testing methodology where the ECU runs with simulated plant models in real-time.

HiPot

High Potential test. Applies high voltage to verify insulation integrity between HV and chassis.

ICT

In-Circuit Test. Automated test of individual components on a PCB using a bed-of-nails fixture.

ISR

Interrupt Service Routine. Handler function executed in response to a hardware interrupt signal.

MIL

Model-in-the-Loop. Testing where the control algorithm runs as a simulation with a simulated plant model.

MISRA-C

Motor Industry Software Reliability Association C coding standard for safety-critical embedded systems.

NTC

Negative Temperature Coefficient thermistor. Temperature sensor whose resistance decreases as temperature rises.

OTA

Over-the-Air. Method of distributing firmware updates wirelessly to embedded devices.

PIL

Processor-in-the-Loop. Testing where compiled code runs on target hardware against a simulated plant model.

RLS

Recursive Least Squares. Adaptive algorithm for real-time parameter identification in battery models.

RTT

Real-Time Transfer. Segger technology for bidirectional data transfer between host and target via debug interface.

SAR

Successive Approximation Register. ADC architecture providing medium speed with low power consumption.

SECDED

Single Error Correction, Double Error Detection. ECC scheme that corrects one-bit and detects two-bit errors.

SOH

State of Health. Percentage indicating remaining battery capacity relative to its nominal new capacity.

SOC

State of Charge. Percentage indicating remaining battery charge relative to current maximum capacity.

SRAM

Static Random Access Memory. Volatile memory used for stack, heap, and global variables during runtime.

SWD

Serial Wire Debug. Two-pin debug interface (clock + data) for ARM Cortex-M microcontrollers.

UDS

Unified Diagnostic Services. ISO 14229 standard for automotive diagnostic communication over CAN.

WCET

Worst-Case Execution Time. Maximum time a task or function takes to execute under any conditions.

About This Guide

This **BMS Firmware Diagnostic Guide** was produced by **Fieldcraft Digital** as a comprehensive reference for engineers developing and maintaining battery management system firmware on the NXP S32K144 platform.

All code examples, timing analyses, and diagnostic procedures have been verified against production BMS firmware running FreeRTOS 10.4 on the S32K144EVB-Q100 evaluation board. Flash budgets, RAM maps, and timing calculations are representative of a 16-cell automotive battery pack application.

For corrections, updates, or technical inquiries, contact Fieldcraft Digital through the support portal referenced on your purchase receipt.

> **Document Information**
>
> Title: BMS Firmware Diagnostic Guide
> Edition: v1.0 (2026)
> Publisher: Fieldcraft Digital
> ISBN: FD-BMS-2026-001
> Classification: Professional Technical Reference
> Revision Date: January 2026

Fieldcraft Digital — Diagnose. Optimize. Deploy.