Debugging Tools and Techniques: Peripheral Register and Memory Inspection (i.MX RT1050-EVKB)

1) Foundations and precise definitions

Peripheral: A hardware block outside the CPU core (for example LPUART, GPIO, I2C, timers) controlled through memory-mapped registers.

Register: A 8/16/32-bit location at a fixed address controlling or reporting peripheral state. Registers are grouped under a *base address*, with offsets for each field (e.g., `BAUD` at +0x10 from LPUART base).

Memory-mapped I/O (MMIO): Peripherals appear in the processor's address space; reading/writing a specific address talks to hardware, not RAM.

Bitfield: A set of bits inside a register controlling a feature (e.g., `LPUART_BAUD_OSR` is the oversampling ratio bitfield inside the LPUART `BAUD` register).

Volatile: A C/C++ type qualifier stating that the value may change at any time outside program control (e.g., a status register). It prevents the compiler from optimizing away accesses. All register structures in the SDK mark fields `__IO`/`__I`/`__O` which expand to `volatile` forms.

Breakpoint: Halts the CPU when execution reaches an address.

Watchpoint (Hardware Data Watchpoint): Halts the CPU when a *memory address* is read/written. Implemented by the Arm Data Watchpoint and Trace (DWT) unit on Cortex-M7. Watchpoints are scarce resources (typically 2–4 comparators).

Endianness: Cortex-M7 in i.MX RT1050 is little-endian. Multi-byte words in RAM and registers are laid out least-significant byte first.

Read side effects: Some registers are *pop-on-read* (e.g., reading a FIFO data register consumes an entry). Some status bits are *write-1-to-clear (W1C)*. Inspecting must respect these semantics.

Caches and coherency: The Cortex-M7 has separate I-Cache and D-Cache. Peripheral regions are non-cacheable by default, but RAM buffers *are* cacheable unless placed in non-cacheable sections or cleaned/invalidated around DMA/EDMA activity. This is critical for trustworthy memory inspection at runtime.

2) The i.MX RT1050 register landscape (addresses you will actually use)

All addresses below are provided by the SDK headers for device MIMXRT1052 (the EVKB uses an RT1052 device variant). The SDK defines base addresses and typed pointers. Examples:

- **LPUART base addresses** (Low-Power UART):
  - `LPUART1_BASE = 0x4018_4000`
  - `LPUART2_BASE = 0x4018_8000`, … up to `LPUART8_BASE = 0x401A_0000`
  - Typed accessors: `#define LPUART1 ((LPUART_Type *)LPUART1_BASE)` … `LPUART8`.
    The `LPUART_Type` structure exposes fields like `BAUD`, `STAT`, `CTRL`, `FIFO`, `WATER` with documented offsets (e.g., `BAUD` at +0x10).
- **GPIO base addresses**:
  - `GPIO1_BASE = 0x401B_8000`, `GPIO2_BASE = 0x401B_C000`, `GPIO3_BASE = 0x401C_0000`, `GPIO4_BASE = 0x401C_4000`, `GPIO5_BASE = 0x400C_0000`
    Typed accessors: `GPIO1`..`GPIO5` of type `GPIO_Type` with registers `DR`, `GDIR`, `PSR`, etc.
- **IOMUXC (IO Multiplexer) base**:
  - `IOMUXC_BASE = 0x401F_8000` plus companion GPR blocks. Pin mux and pad control are set via SDK functions like `IOMUXC_SetPinMux()` and `IOMUXC_SetPinConfig()` and can be inspected via memory view when needed.

  These definitions come straight from the SDK headers (`devices/MIMXRT1052/MIMXRT1052_COMMON.h` and `devices/MIMXRT1052/periph/PERI_*.h`), ensuring your code and your debugger agree on addresses and register layouts.

A quick look at the LPUART register struct

```
// devices/MIMXRT1052/periph/PERI_LPUART.h (excerpt)
typedef struct {
  __I  uint32_t VERID;  // 0x00
  __I  uint32_t PARAM;  // 0x04
  __IO uint32_t GLOBAL; // 0x08
  __IO uint32_t PINCFG; // 0x0C
  __IO uint32_t BAUD;   // 0x10
  __IO uint32_t STAT;   // 0x14
  __IO uint32_t CTRL;   // 0x18
  __IO uint32_t DATA;   // 0x1C  (pop-on-read)
  __IO uint32_t MATCH;  // 0x20
  __IO uint32_t MODIR;  // 0x24
  __IO uint32_t FIFO;   // 0x28
  __IO uint32_t WATER;  // 0x2C
} LPUART_Type;
```

3) Tooling you will use on EVKB-i.MXRT1050

MCUXpresso IDE (or your GDB-based IDE)

- **Peripheral Registers view**: presents each module (LPUART, GPIO, IOMUXC, CCM) with human-readable fields and bit descriptions. Safe to *read*. Be careful when editing W1C or pop-on-read fields.
- **Memory view**: view any address as 8/16/32-bit values or ASCII. Useful for RAM buffers, stacks, and also for peeking at register blocks when a peripheral is not in the Peripheral view.
- **Live Expressions / Expressions**: track variables and *addresses* that update while running, with minimal intrusiveness. For side-effect registers, prefer periodic halts or snapshots.

GDB essentials (works under MCUXpresso, VS Code, or CLI)

- Inspect memory/registers:
  - `x/16wx 0x40184000` — dump 16 words from LPUART1 base.
  - `p/x LPUART1->BAUD` — print BAUD register in hex.
  - `p/t ((LPUART_Type*)0x40184000)->STAT` — print STAT bits in binary.
- Safe writes (use sparingly):
  - `set {volatile unsigned long}0x401F8010 = 0x...` — write a mux register you *fully* understand.
- Watchpoints (halt on data access):
  - `watch *0x20200000` — halt when your ring buffer head changes.

  - Use hardware watchpoints (`hwatch`) for non-RAM addresses or to ensure non-intrusive behavior.

Trace and non-intrusive observation

- **DWT/ITM**: Data Watchpoint and Trace plus Instrumentation Trace Macrocell allow non-intrusive event reporting (printf-like through SWO) and watchpoints with minimal latency. The EVKB brings out SWD; SWO availability depends on your probe and project wiring. Use for timing stamps without perturbing UART activity.

4) Reading and writing registers from C (two idioms)

Idiom A: typed peripheral structures (preferred)

```c
#include "fsl_device_registers.h"    // pulls in MIMXRT1052 headers
#include "fsl_lpuart.h"             // driver API
```

```
static void dump_lpuart1(void) {
    volatile LPUART_Type *u = LPUART1; // 0x40184000
    uint32_t baud  = u->BAUD;
    uint32_t stat  = u->STAT;
    uint32_t ctrl  = u->CTRL;
    uint32_t fifo  = u->FIFO;
    uint32_t water = u->WATER;
    (void)baud; (void)stat; (void)ctrl; (void)fifo; (void)water; // set break
here
}
```

Pause at the comment and inspect `baud/stat/ctrl/...` in your debugger. This is safe and readable.

Idiom B: raw MMIO with `volatile` pointers (when no struct is available)

```
#define REG32(addr) (*(volatile uint32_t *)(addr))
#define LPUART1_BAUD_ADDR  (0x40184000u + 0x10u)

uint32_t baud = REG32(LPUART1_BAUD_ADDR);
```

This is useful for quick experiments or when you only have an address/offset from the reference manual.

> **Never** access registers without `volatile`. The compiler is allowed to cache non-volatile reads/writes and you will be debugging an optimization artifact instead of hardware.

---

5) Generic lab: Inspecting LPUART1 and verifying a known configuration

Goal: load the SDK's LPUART interrupt example, halt at init, and prove (by registers) that the peripheral is configured for the desired baud, parity, stop bits, and FIFO watermarks.

Project to open:
`boards/evkbimxrt1050/driver_examples/lpuart/interrupt_transfer/`

Pins in use (from its `pin_mux.c`): `GPIO_AD_B0_12` as `LPUART1_TXD` and `GPIO_AD_B0_13` as `LPUART1_RXD` configured via `IOMUXC_SetPinMux()` and `IOMUXC_SetPinConfig()`.

Step-by-step 1. Build and flash the example. Connect a USB-UART dongle (115200-8-N-1) to the EVKB UART header that routes LPUART1. 2. Open the source and set a breakpoint right after `LPUART_Init()` in `main()`. 3. Run to the breakpoint. Open **Peripheral Registers → LPUART1**. Also open **Memory** at `0x40184000` with 32-bit words. 4. Check these fields: - **BAUD**: `OSR` should typically be 15 (meaning 16× oversampling). `SBR` should be `round(Clock/(Baud*(OSR+1)))`. - With a typical UART clock of 80 MHz (PLL3/6) and 115200 baud, `SBR ≈ 80e6 / (115200 * 16) ≈ 43` (0x2B). - **CTRL**: confirm `TE` (Transmitter Enable) and `RE` (Receiver Enable) bits are 1; parity bits off unless example enables them. - **FIFO/WATER**: transmit/receive FIFO sizes and watermark levels set by the driver. 5.

Confirm **IOMUXC** pin mux: open **Memory** at `IOMUXC_BASE` and navigate to the mux registers for `GPIO_AD_B0_12/13`. The example called: c `IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_12_LPUART1_TXD, 0U);` `IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_13_LPUART1_RXD, 0U);` You should see ALT value bits matching LPUART1 on those pads, and reasonable drive strength/slew (`SetPinConfig` argument `0x10B0U`).

GDB one-liners (equivalents):

```
p/x LPUART1->BAUD
p/x LPUART1->CTRL
p/x LPUART1->STAT
x/8wx 0x40184010     # BAUD onward
```

Success criteria: The register window matches the expected configuration and the example echoes characters reliably with the expected watermarks and without overrun (`OR`) or framing errors (`FE`) in `STAT`.

---

6) Memory inspection you will actually use

Where things live on i.MX RT1050 - **Instruction TCM (ITCM)** at `0x0000_0000` (tightly coupled, non-cacheable, fastest code). - **Data TCM (DTCM)** around `0x2000_0000` (tightly coupled RAM for stacks/ISRs; non-cacheable, deterministic). - **On-chip RAM (OCRAM)** around `0x2020_0000` (cacheable by default; larger, good for buffers and heaps). - **External memory** (FlexSPI NOR, SDRAM if fitted) — cacheable; requires extra care with DMA/coherency.

Practical tips - **Stacks**: Use the Memory view to watch the main/ISR stacks grow. Add watchpoints on high-water marks during stress tests. - **Ring buffers**: For UART/EDMA, place receive buffers in non-cacheable RAM to guarantee the debugger sees fresh data. In SDK projects use the attributes shown below.

SDK attributes for non-cacheable buffers

```c
#include "fsl_cache.h"   // brings in cache helpers and attributes

#define ECHO_BUFFER_LENGTH 256
AT_NONCACHEABLE_SECTION_INIT(uint8_t g_rx[ECHO_BUFFER_LENGTH]) = {0};
AT_NONCACHEABLE_SECTION_INIT(uint8_t g_tx[ECHO_BUFFER_LENGTH]) = {0};
```

If you *must* use cacheable RAM with DMA, bracket transfers with cache maintenance:

```c
SCB_CleanDCache_by_Addr((uint32_t *)g_tx, ECHO_BUFFER_LENGTH);
SCB_InvalidateDCache_by_Addr((uint32_t *)g_rx, ECHO_BUFFER_LENGTH);
```

---

7) Avionics use case lab: Triaging ARINC 429 ingest via ADK-8582 over UART

Scenario: In an aircraft integration rig, an ARINC 429 receiver/adapter board (ADK-8582) streams decoded 32-bit ARINC 429 words (label, Source/Destination Identifier, data, Sign/Status Matrix, parity) as framed ASCII/Binary messages over UART to the EVKB-i.MXRT1050. The flight test team reports intermittent "label mismatch" and dropped words when two Line Replaceable Units (LRUs) talk at high rate.

Goal: Use *register and memory inspection only* to decide whether the fault is in the UART path on i.MX RT1050 (framing/overrun/config) or upstream (ADK-8582/line wiring), and to confirm ring-buffer pressure.

Hardware - Connect ADK-8582 UART TX/RX to EVKB `LPUART1` pins: `GPIO_AD_B0_12` (TXD), `GPIO_AD_B0_13` (RXD). Common ground. - Set ADK-8582 UART parameters to match the EVKB project (e.g., 115200-8-N-1). Use the rig's Interface Control Document (ICD) for the exact framing format and label list.

Firmware (based on SDK LPUART driver)

```c
#include "fsl_lpuart.h"
#include "fsl_iomuxc.h"
#include "fsl_cache.h"

#define RX_BUF_SZ 1024
AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t s_rxBuf[RX_BUF_SZ], 16);
static volatile size_t s_head = 0, s_tail = 0; // ring indices (watch these!)

static void uart1_init(uint32_t srcClkHz, uint32_t baud)
{
    lpuart_config_t cfg;
    LPUART_GetDefaultConfig(&cfg);
    cfg.baudRate_Bps = baud;              // per ICD (e.g., 115200)
    cfg.enableTx = true;
    cfg.enableRx = true;

    CLOCK_EnableClock(kCLOCK_Iomuxc);
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_12_LPUART1_TXD, 0U);
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_13_LPUART1_RXD, 0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_12_LPUART1_TXD, 0x10B0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_13_LPUART1_RXD, 0x10B0U);

    LPUART_Init(LPUART1, &cfg, srcClkHz);
    LPUART_EnableInterrupts(LPUART1, kLPUART_RxDataRegFullInterruptEnable |
kLPUART_RxOverrunInterruptEnable);
    EnableIRQ(LPUART1_IRQn);
}
```

```
void LPUART1_IRQHandler(void)
{
    uint32_t stat = LPUART_GetStatusFlags(LPUART1);
    if (stat & kLPUART_RxOverrunFlag) {
        LPUART_ClearStatusFlags(LPUART1, kLPUART_RxOverrunFlag);
        // optional: increment a diagnostic counter you can watch in the
debugger
    }
    if (stat & kLPUART_RxDataRegFullFlag) {
        uint8_t d = LPUART_ReadByte(LPUART1); // pops DATA (side effect!)
        size_t n = (s_head + 1U) % RX_BUF_SZ;
        if (n != s_tail) { s_rxBuf[s_head] = d; s_head = n; }
        else { /* overflow: drop or mark */ }
    }
    __DSB(); // ensure all writes complete before exit
}
```

What to inspect 1. **UART health** via registers: - `LPUART1->STAT`: watch `FE` (framing error), `NF` (noise), `OR` (overrun). None should accumulate. Use the Peripheral view and also `p/x LPUART1->STAT` repeatedly while the rig runs. - `LPUART1->BAUD`: confirm the configured `SBR` and `OSR` match the intended baud. - `LPUART1->WATER`/`FIFO`: observe `RXCOUNT` growth and watermark behavior under load. 2. **Buffer pressure** in RAM: - Add watchpoints on `s_head` and `s_tail` addresses. In GDB: `watch s_head`. - In **Memory view**, point at `s_rxBuf` and verify bytes change continuously (non-cacheable buffer ensures you see fresh data without manual cache ops). 3. **If overruns appear** but timestamps show the CPU is not starved: - Increase `WATER` RX threshold to trigger ISR earlier, or move to EDMA and repeat with `AT_NONCACHEABLE_SECTION` buffers. - If overruns persist with generous ISR, suspect upstream framing—use a logic analyzer on RX or the ADK-8582 diagnostics per ICD.

Decision point - If `STAT.OR` increments with `RXCOUNT` pinned near full → **firmware buffering/latency** issue. - If `STAT.FE` spikes with no ring pressure → **upstream signal integrity or framing** issue. - If `BAUD` mismatch discovered → **configuration** issue (wrong UART clock mux/divider or wrong baud).

---

8) Advanced inspection techniques (Cortex-M7 on i.MX RT)

- **Hardware watchpoints on MMIO**: You can watch a *RAM* address easily. Watching MMIO (peripheral) reads/writes is probe/IDE-dependent and may be intrusive. Prefer counting errors in RAM variables and watching those.
- **Freeze behavior while halted**: When you hit a breakpoint, the core halts but most peripheral clocks keep running unless explicitly frozen. Avoid halting in the middle of a time-critical ISR; prefer periodic snapshots or DWT counters. Disable watchdogs during lab work or use the SDK examples that already handle them.
- **Cache coherency**: For DMA/EDMA peripherals, either use non-cacheable sections (`AT_NONCACHEABLE_SECTION[_ALIGN/INIT]`) or call `SCB_CleanDCache_by_Addr` /

SCB_InvalidateDCache_by_Addr around producer/consumer edges. Inspecting a cached buffer without invalidation shows *stale* data.

- **Clock gating check**: If a peripheral seems unresponsive, inspect the Clock Control Module (CCM) CCGRx gates via the Peripheral view, or (preferably) call the SDK clock API (CLOCK_EnableClock(...)) and re-read the target registers to confirm.

---

9) Best practices (avionics-grade)

1. **Observe before you change**: Prefer read-only inspection first. If you must poke a register, do it from code you control (with comments), not ad-hoc from a debugger.
2. **Respect side effects**: Know which registers are pop-on-read or W1C. Reading a data FIFO to "look around" may drop a byte and create the very fault you are chasing.
3. **Make buffer visibility deterministic**: Use non-cacheable buffers or cache maintenance so what you *see* in the debugger is what the DMA/ISR just wrote.
4. **Instrument with counters**: Maintain rx_overrun_count, framing_error_count, and max_ring_depth. Watch those variables (not raw MMIO) during long runs.
5. **Pin mux first**: If a peripheral is "dead," verify IOMUXC mux/pad registers *and* CCM clocking before touching the peripheral block.
6. **Use typed access**: Prefer the SDK's *_Type structs and driver APIs. You get correct offsets and safer bit masks.
7. **Keep probes non-intrusive**: Avoid printf() in hot paths. Use ITM (if available) or snapshot variables and read them post-run.
8. **Lock configurations**: For repeatable tests (important for Design Assurance Level (DAL) A/B contexts), capture the exact clock tree and baud calculations in logs.

---

10) Common failure patterns and quick triage

- **Nothing happens**: Wrong IOMUXC function or peripheral clock gate closed. Inspect IOMUXC mux registers and CCM CCGRx. Use CLOCK_GetFreq() to confirm source clock.
- **UART drops bytes under load**: RX watermark too high; ISR latency; ring too small; cacheable buffer with DMA; D-Cache not invalidated.
- **Data looks "scrambled"**: Endianness misinterpretation when decoding multi-byte fields; mixing ASCII and binary; reading DATA while EDMA also owns the FIFO.
- **Register reads are "frozen"**: You are reading cached RAM, not MMIO; or you are inspecting a shadow variable that never updates because it isn't volatile.

---

11) Exercises with solutions (using the provided SDK)

Exercise A — GPIO sanity check by register inspection

Task: Toggle an LED on the EVKB by writing GPIO1 registers and prove via register reads that `GDIR` and `DR` are correct.

Steps 1. Create a bare project or reuse any example. Enable clock to IOMUXC and set the desired pad to `GPIO1_IO09` (for example, an LED line on your board revision—check EVKB schematic). Configure pad drive strength. 2. Set `GPIO1->GDIR |= (1U << 9)` and toggle `GPIO1->DR ^= (1U << 9)` in a loop with a delay. 3. In the debugger, watch `GPIO1->PSR` to see the pin state, and inspect `DR`/`GDIR`.

Solution snippet

```c
#include "fsl_iomuxc.h"
#include "fsl_common.h"

static void gpio_led_init(void)
{
    CLOCK_EnableClock(kCLOCK_Iomuxc);
    IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_09_GPIO1_IO09, 0U);
    IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B0_09_GPIO1_IO09, 0x10B0U);
    GPIO1->GDIR |= (1u << 9);
}

static void gpio_led_toggle(void)
{
    GPIO1->DR ^= (1u << 9);
}
```

Inspect: `p/x GPIO1->GDIR` (bit 9 = 1), `p/x GPIO1->DR`, `p/x GPIO1->PSR`.

Exercise B — LPUART register math you can verify

Task: Prove that your configured baud is actually what LPUART will generate by inspecting `BAUD` and re-computing.

Steps 1. In the SDK LPUART interrupt example, set `cfg.baudRate_Bps = 115200;` and ensure the UART clock is 80 MHz (default PLL3/6 path). 2. Halt after `LPUART_Init()`. 3. Read `LPUART1->BAUD` and decode `OSR` and `SBR` fields.

What you should see - With `OSR = 15` (i.e., 16× oversampling) and `SBR = 43 (0x2B)`, the effective baud is: `baud = src_clk / ((OSR + 1) * SBR) = 80,000,000 / (16 * 43) ≈ 116,279` baud (driver may adjust OSR/SBR to minimize error). - If your project uses a different UART clock path or divider, repeat the math with the actual `CLOCK_GetFreq(kCLOCK_UartClk)` value.

Inspect: `p/x LPUART1->BAUD`, `p/x LPUART1->CTRL` (TE/RE bits set), `p/x LPUART1->WATER` (`RXCOUNT/TXCOUNT`).

Exercise C — ARINC 429 ring-buffer overflow hunt with watchpoints

Task: While streaming ARINC 429 messages from the ADK-8582 at high rate, prove whether ring overflow happens and whether UART overruns occur.

Steps 1. Use the ISR-based code in Section 7 (or convert to EDMA if you need even higher throughput). 2. Set a hardware watchpoint on s_head and `s_tail`. In GDB: `watch s_head`, `watch s_tail`. 3. Add counters: `volatile uint32_t rx_or_count; volatile uint32_t fe_count;` and increment them in the ISR when flags are seen. 4. Run for 2 minutes under peak rig traffic.

Inspect and decide - If `rx_or_count > 0` with s_head roughly equal to `s_tail` most of the time → overrun likely at the UART (ISR latency/watermark). - If `fe_count` climbs but `rx_or_count == 0` and ring remains light → upstream framing/line issue. - If s_head repeatedly laps `s_tail` → ring too small; increase `RX_BUF_SZ` and/or move to EDMA; re-run test.

Bonus: Move the buffer into DTCM (non-cacheable, deterministic) and confirm that observed memory contents track ISR writes exactly.

---

12) Specific avionics cross-checks you should remember

- **ICD first**: For any ARINC 429 label/SDI/SSM interpretation, use the integration rig's Interface Control Document. Your debugger confirms byte flow and timing; semantic meaning comes from the ICD.
- **DAL-aware logging**: Capture register snapshots and buffer depth signatures for traceability—store them alongside test artifacts to support DO-178C objectives.
- **Electrical sanity**: If you see intermittent framing errors with stable CPU load, use a scope/LA on the UART RX line to rule out cabling or common-mode issues before changing software.

Appendix — Handy addresses and masks (from the SDK)

- `LPUART1_BASE = 0x40184000`, `LPUART1` typed pointer → fields `BAUD (0x10)`, `STAT (0x14)`, `CTRL (0x18)`, `FIFO (0x28)`, `WATER (0x2C)`.
- `GPIO1_BASE = 0x401B8000` → `DR`, `GDIR`, `PSR`.
- `IOMUXC_BASE = 0x401F8000` → use `IOMUXC_SetPinMux/SetPinConfig` APIs to avoid bit-banging mux registers.

- Non-cacheable buffer attributes: `AT_NONCACHEABLE_SECTION`, `AT_NONCACHEABLE_SECTION_INIT`, `AT_NONCACHEABLE_SECTION_ALIGN` (see multiple SDK examples under `boards/evkbimxrt1050/...`).

Summary

By combining *peripheral register* inspection (for ground truth about what the silicon is doing) and *RAM inspection* (for ground truth about what your firmware is buffering), you can quickly isolate whether an issue is a configuration defect, a software latency problem, or an upstream data quality problem. These techniques are mandatory for deterministic, certifiable avionics firmware on the i.MX RT1050 family.