Technical Laboratory Report: Interrupt Latency Measurement on the i.MX RT1052 Microcontroller

1. Objective

This report aims to measure and compare the interrupt latency using three different implementation approaches on the i.MX RT1052 microcontroller: bare metal code, FreeRTOS operating system, and external GPIO trigger. The results are compared with those from NXP's Application Note AN12078【1】. The goal is to practice interrupt latency analysis with practical tools such as an oscilloscope and disassembly debugging—fundamental concepts in microcontroller programming.

---

2. Configuration

2.1 Required Tools

- **Development board**: MIMXRT1050-EVKB
- **IDE:** MCUXpresso IDE v24.12 Eclipse
- **Oscilloscope:** Agilent Infiniium DSA90254A (20 GSa/s)

2.2 Build and Flash

1. Clone the repository: https://github.com/jvmreis/nxp_interrupt_lesson.git
2. Open MCUXpresso IDE
3. Go to `Window > Show View > Other` and enable the Disassembly tab
4. Go to `File > Import Project from File System`
5. Select the cloned project directory, e.g., `bar_metal` or `free_rtos`
6. Click **Build** to compile
7. Click **debug** to flash the firmware

2.3 Project Configuration Overview

2.3.1 Debug GPIO

The interrupt response is observable via a debug signal on GPIO1_IO19, routed to header J22, pin 8. This allows precise measurement of ISR execution with the oscilloscope.

2.3.2 Slew Rate

All GPIOs were set to "Fast" slew rate using MCUXpresso Config Tools. This reduces output capacitance and improves timing accuracy for signal transitions.

2.3.3 Button

The SW8 button was used for external interrupts, configured on GPIO5_IO00 with a falling-edge trigger and internal pull-up. It triggers `GPIO5_Combined_0_15_IRQHandler`.

2.3.4 Clock

The system's main clock was configured at 600 MHz, used as the base frequency to calculate CPU cycles from measured times.

2.3.5 Internal Timer Interrupt

The GPT2_COMPARE3 signal was routed to TP11 (GPIO_AD_B0_08)【2】for monitoring during internal interrupt tests.

2.3 Enabling the Disassembly Debugger

To add the disassembly debugger view and analyze instructions at the assembly level during debugging:

1. In the **top-right corner** of the MCUXpresso IDE, click on **`Window`**.
2. Select **`Show View`** from the dropdown menu.
3. Click on **`Other...`** to open the full list of available views.
4. In the dialog that appears, type or search for **`Disassembly`**.
5. Select **`Disassembly`** under the **Debug** category and click **OK**.

The Disassembly tab will now appear, allowing you to inspect and step through instructions during breakpoints and interrupts. This is essential for low-level timing analysis and optimizing ISR performance.

---

3. Methodology

The experiment evaluates interrupt latency in three scenarios:

1. Bare Metal with GPT2
2. FreeRTOS with semaphore and context switch
3. External GPIO interrupt (button)

The time between the interrupt trigger and the toggle on GPIO1_IO19 is measured using the oscilloscope. Measurements t_1 and t_2 (in ns) are converted to clock cycles based on a 600 MHz frequency.

3.1 Bare Metal (GPT2)

The diagram below presents a schematic representation of the experiment, in which the GPT2_COMPARE3 comparator signal acts as the interrupt event. The signal generated by the ISR is represented by a second trace, which demonstrates the delay between the generation of the event and the system's response. This diagram is essential for visually understanding how the oscilloscope was used to capture the moments t_1 (time between the event and the response) and t_2 (duration of the ISR execution). The GPT2_COMPARE3 channel was configured in toggle mode, generating a periodic interrupt every 57 seconds. When triggered, the ISR directly writes to the GPIO1 data register, toggling the logical value on pin IO19 [3]. This results in a visible pulse on the oscilloscope.

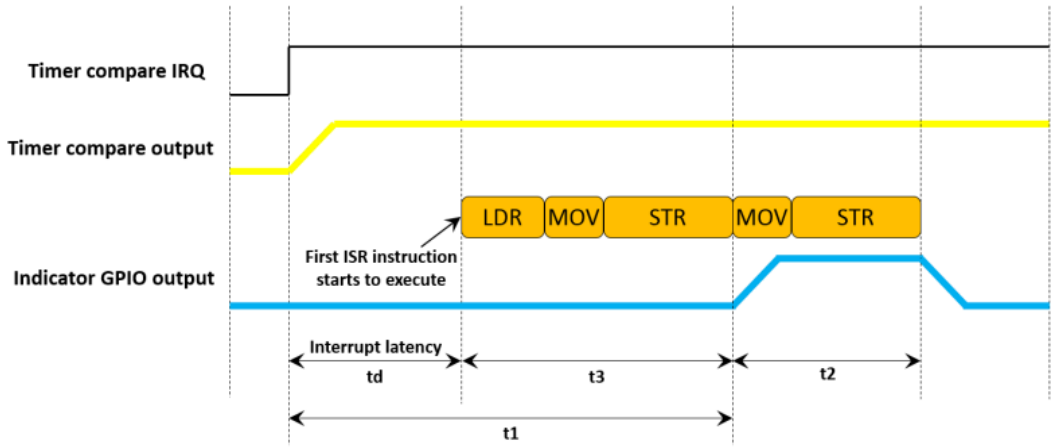

Figure 1 – Bare-Metal Latency Measurement Diagram【1】

Steps:

1. Connect oscilloscope channel 1 to TP11 (event) and channel 2 to pin J22 (debug).
2. Set the trigger to the falling edge on channel 1.
3. Use Single mode, wait, and measure t_1 and t_2.

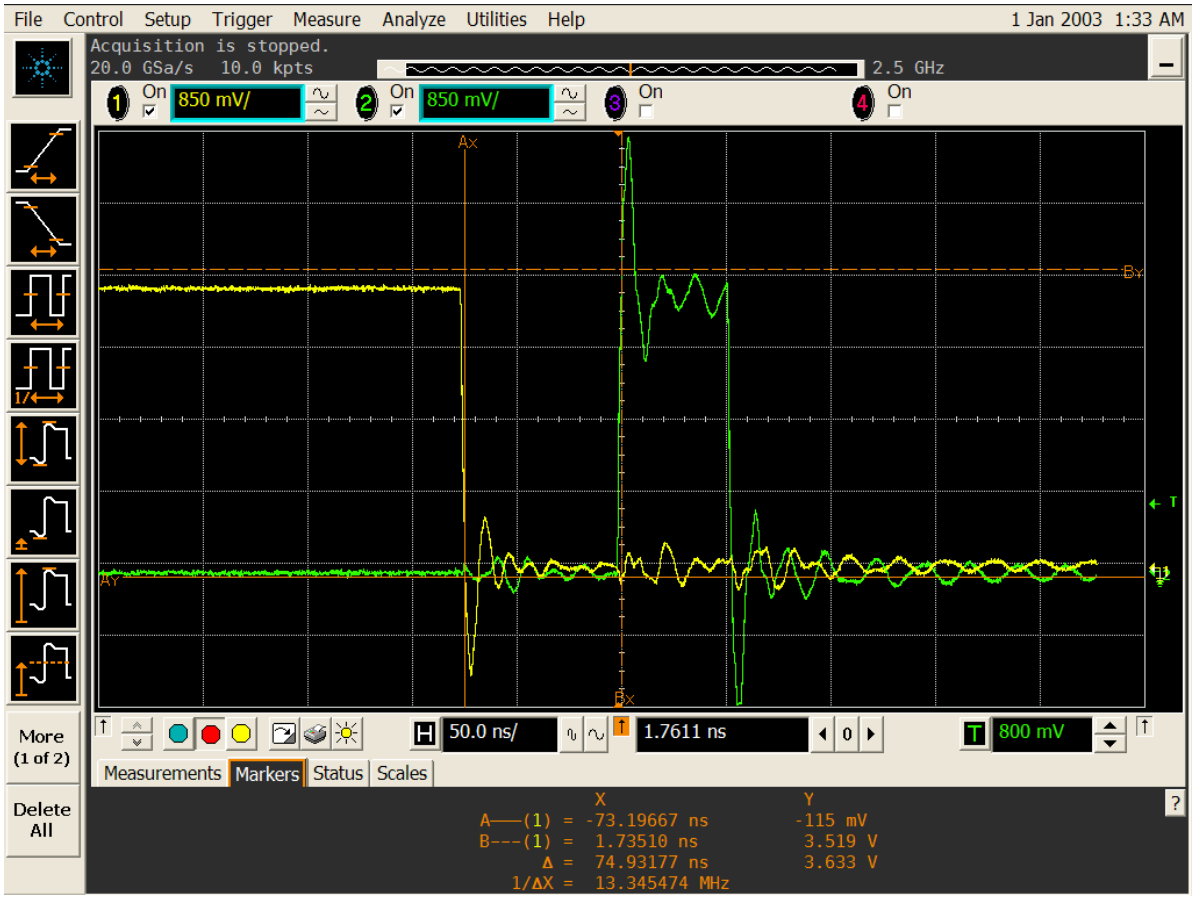

Figure 2 – Latency Measurement of Interrupt T1 with GPT2 (Bare Metal):

This image shows two overlapping signals: channel 1 (yellow) represents the event generated by GPT2_COMPARE3, while channel 2 (green) shows the toggle of the debug pin (GPIO1_IO19). The distance between the falling and rising edges allows for measurement of the total ISR latency.

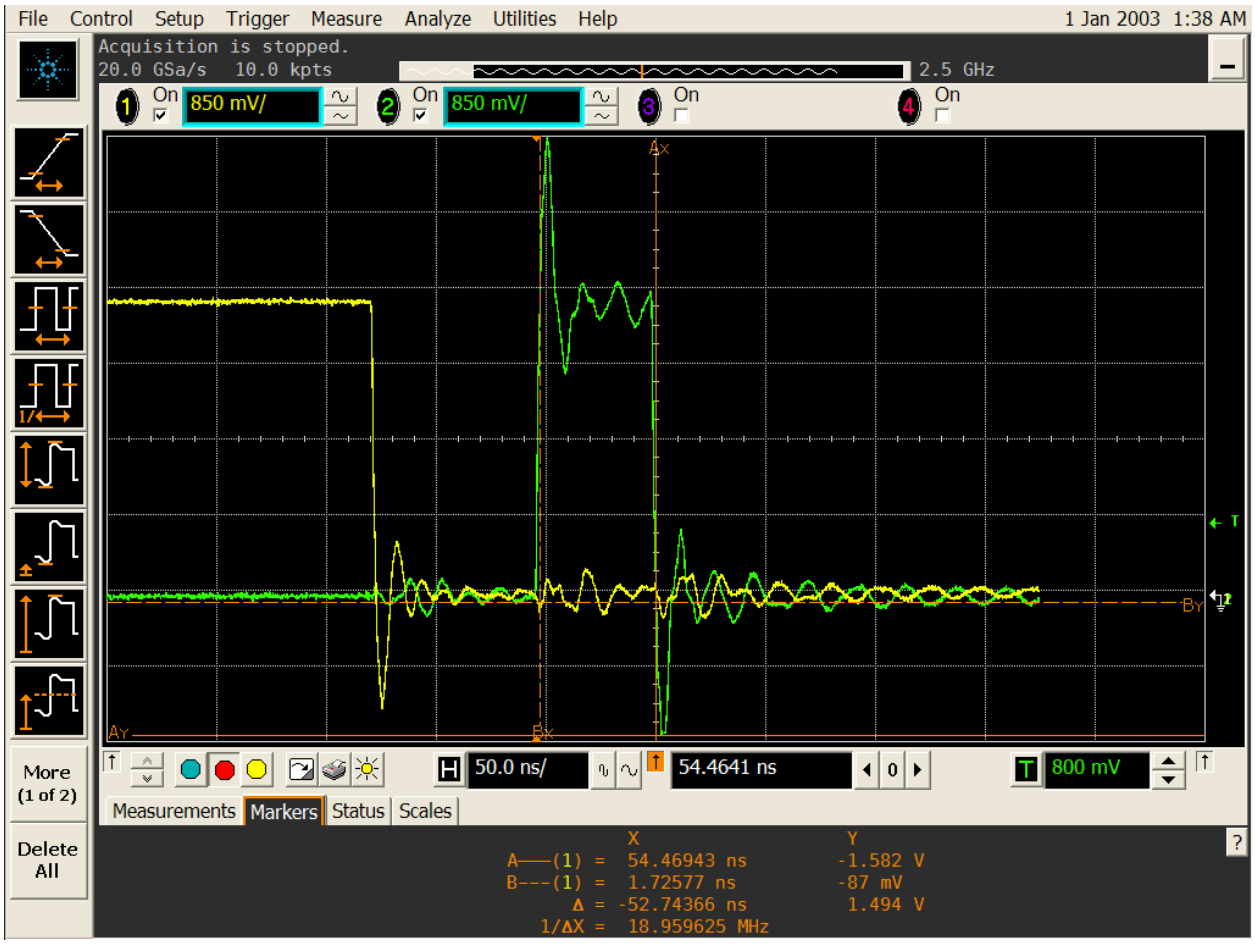

Figure 3 – Latency Measurement of Interrupt T2 with GPT2 (Bare Metal):

4. In debug mode, set a breakpoint in the `GPT2_IRQHandler` function.
5. In the disassembly view, identify how many cycles are used within the ISR.

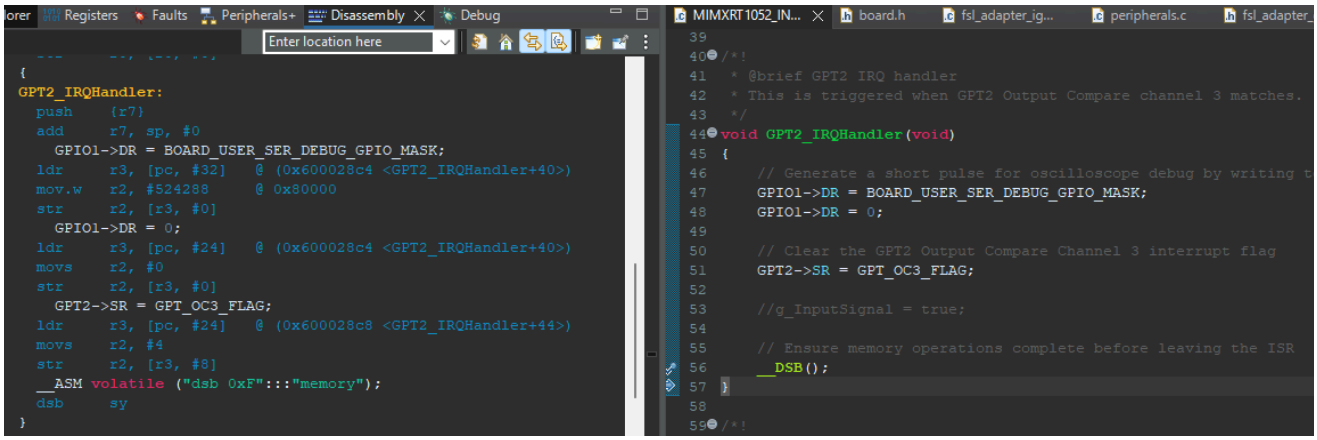

Figure 4 – Disassembly Snippet of the ISR:

This figure shows the assembly-level view of the `GPT2_IRQHandler` function, obtained via the “Disassembly” tab in MCUXpresso IDE. Each listed instruction corresponds to an ARM operation that directly affects the timing of the interrupt response. Memory access instructions (LDR and STR) are particularly important, as their latency can vary depending on the bus used. This analysis is essential to theoretically validate the times observed on the oscilloscope and understand the timing granularity in a bare-metal environment.

Example Instruction Sequence (from Application Note AN12078):

```
LDR.N   R0, [PC, #0x78]   ; GPIO2->DR

MOV.W   R1, #8388608      ; (1 << 23)

STR     R1, [R0]          ; set high

MOVS    R1, #0            ; clear

STR     R1, [R0]          ; set low
```

Cycle Explanation (based on AN12078):

- LDR: 2 cycles (accessing GPIO address)
- MOV.W: 1 cycle (loading immediate value)
- STR: variable (depends on peripheral bus)
- MOVS: 1 cycle
- STR: variable again

STR accesses can vary in cycle count due to bus latency. This is one of the reasons we use the oscilloscope to precisely measure the moment when the GPIO signal changes.

Example Measurement:

- t_1 = 74.66667 ns × 0.6 = 45 cycles
- t_2 = 52.74366 ns × 0.6 = 32 cycles
- t_3 = 2 + t_2 = 34 cycles
- td = 45 – 34 = 11 cycles

3.2 External GPIO Interrupt

The diagram below shows the architecture of the test performed for external interrupts triggered by a button. In this setup, button SW8 acts as the event generator, connected to pin GPIO5_IO00. When pressed, the button triggers an interrupt configured to detect a falling edge. The response to the interrupt is recorded by a pulse on GPIO1_IO19. Oscilloscope channel 1 monitors the button signal, while channel 2 captures the debug pulse on the output GPIO.

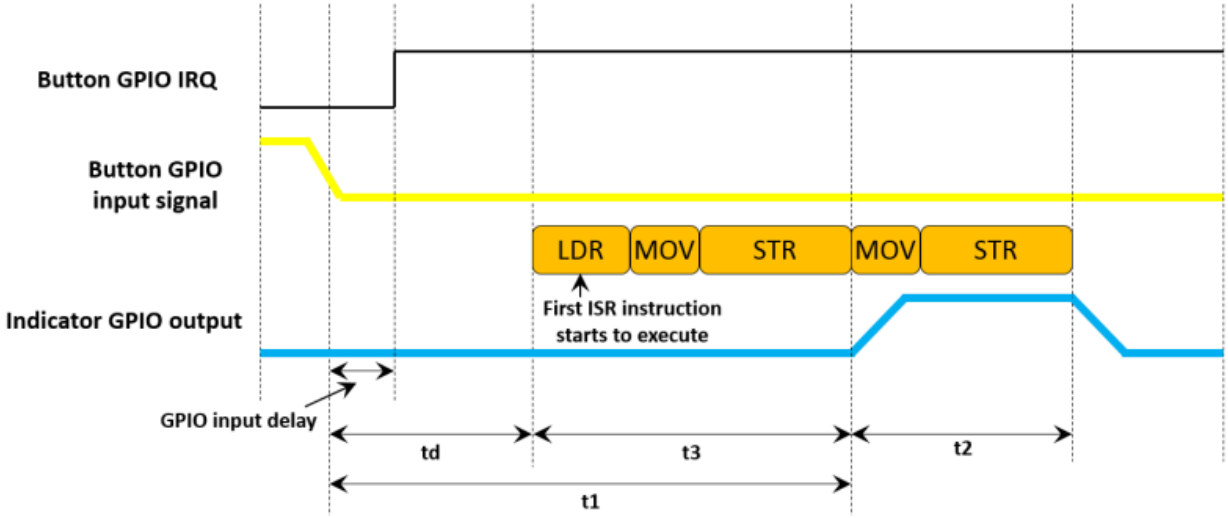

Figure 5 – External GPIO Measurement Diagram【1】

The interval between the falling edge on channel 1 and the rising edge on channel 2 corresponds to the interrupt latency, and the width of the debug pulse represents the ISR execution time. At this stage, we analyze the response time to interrupts generated externally via button SW8, connected to pin GPIO5_IO00. Two distinct approaches were evaluated for handling the interrupt:

- **Direct Mode (faster):** The ISR directly handles the interrupt.
- **Pin Comparison Mode (slower):** The ISR uses the function `HAL_GpioInterruptHandle()`, which processes multiple pins and directs execution to a specific callback.

How to switch between test modes

In the project's source code, there is a preprocessor macro called:

#define `ISR_GPIO_HAL_COMPARATION 0`

This macro defines which approach will be used in the experiment:

- `ISR_GPIO_HAL_COMPARATION == 0` → Direct mode, no pin comparison.

- `ISR_GPIO_HAL_COMPARATION > 0` → Mode using HAL and callback, simulating multiple pins.

To switch between modes:

1. Open the file `MIMXRT1052_INT_lesson.c`.

2. Locate `#define ISR_GPIO_HAL_COMPARATION`.

3. Change the value from 0 to 1 (or vice versa), depending on the desired test mode.

4. Recompile and reflash the firmware to the board.

Step-by-Step Guide to Execute the Experiment

Part 1 – Direct Mode (no pin comparison)

- Set `#define ISR_GPIO_HAL_COMPARATION 0` at the beginning of the code.
- Compile and flash the firmware.
- Connect:
    - Oscilloscope channel 1 to the SW8 button (event).
    - Channel 2 to the debug pin GPIO1_IO19 (J22, pin 8).

- Press the button and capture the response using a falling-edge trigger.
- Measure the times:
    - $\mathbf{t_1}$ (between the button's falling edge and the rising edge of the GPIO signal)
    - $\mathbf{t_2}$ (pulse duration)

- Calculate latency:
    - $\mathbf{t_3}$ **= 2 cycles (LDR) +** $\mathbf{t_2}$
    - **td =** $\mathbf{t_1}$ **-** $\mathbf{t_3}$

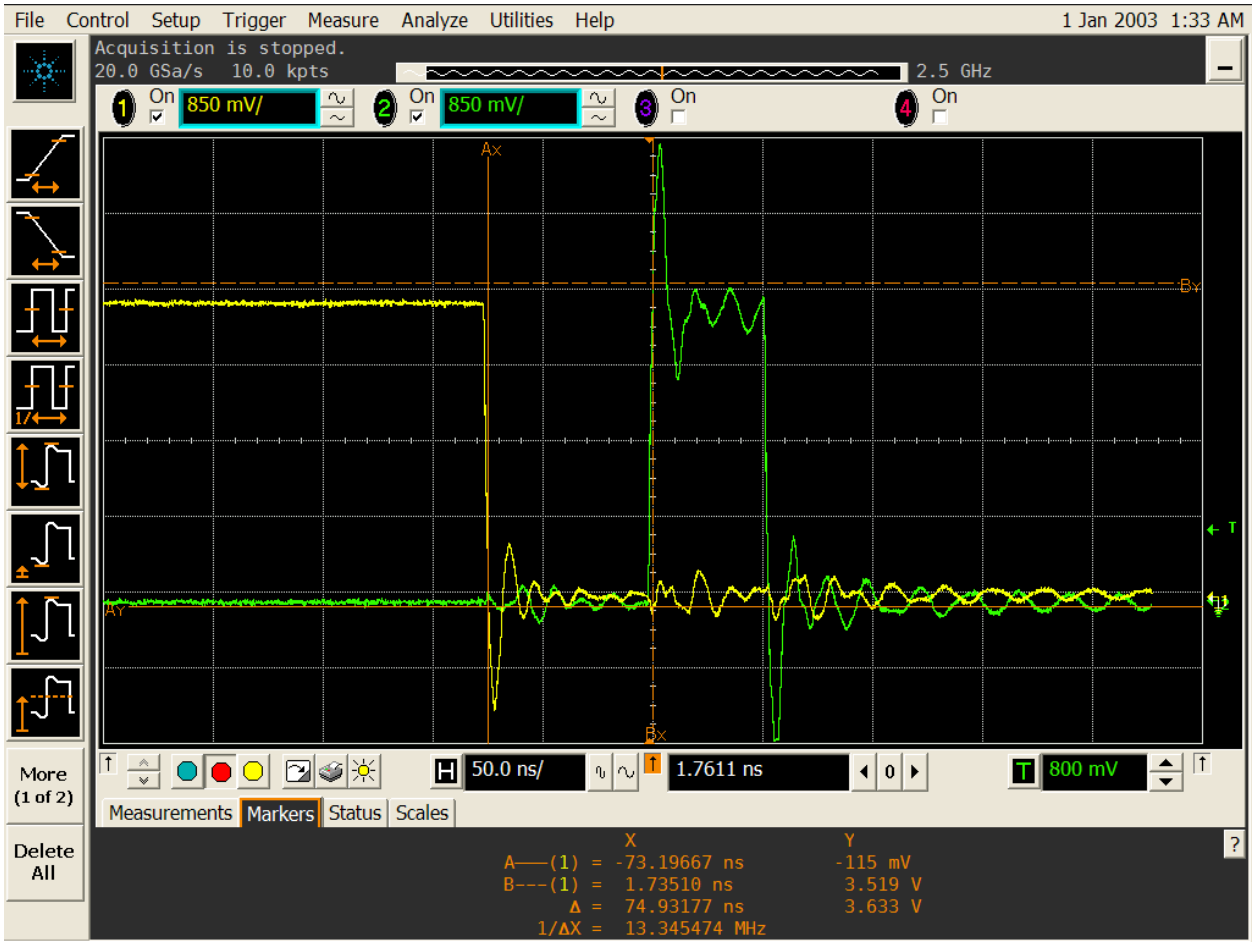

Figure 6 – External Interrupt without Pin Comparison:

In this image, we observe that the time between the button press (channel 1) and the response on the debug pin (channel 2) is relatively short. The ISR responds directly upon detecting the falling edge on the pin, with a response time similar to the bare-metal experiment.

Part 2 – HAL and Callback Mode (with pin comparison)

- Change `#define ISR_GPIO_HAL_COMPARATION` to value `1`.
- Compile and reflash the firmware.
- Repeat the oscilloscope setup as before.
- Press the SW8 button and capture the signals again.

5. Notice that the pulse takes longer to appear due to the overhead of the `HAL_GpioInterruptHandle()` function.

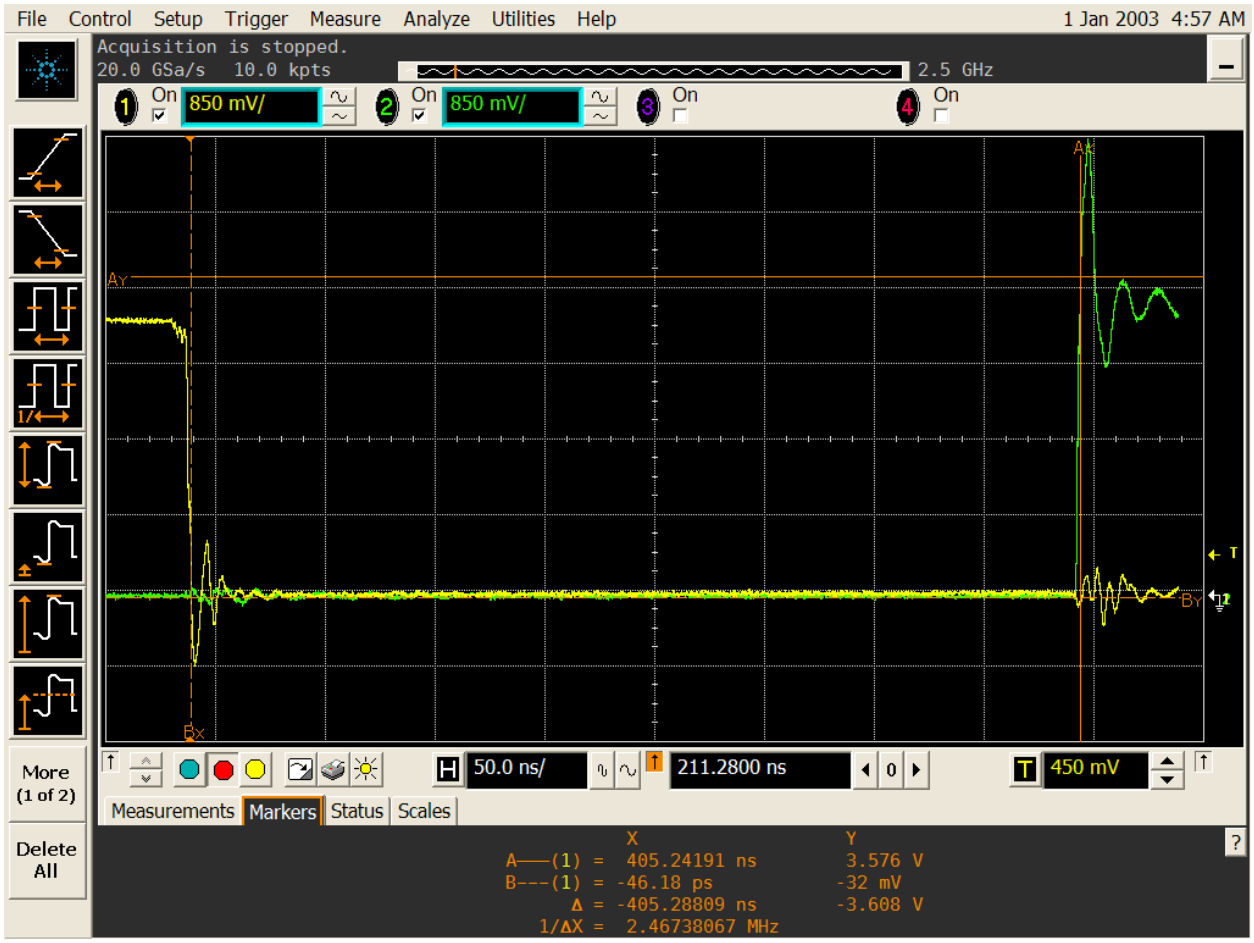

Figure 7 – External Interrupt with Pin Comparison (via HAL):

In this capture, the observed latency is significantly higher. This is due to the overhead of the `HAL_GpioInterruptHandle()` function, which performs additional operations such as reading flags, comparing pins, and calling callbacks.

6. In debug mode, place a breakpoint in the function `BOARD_INITPINS_USER_BUTTON_callback`.
7. In the disassembly view, identify how many cycles are used within the `GPIO05_Combined_0_5_IRQHandler` ISR.

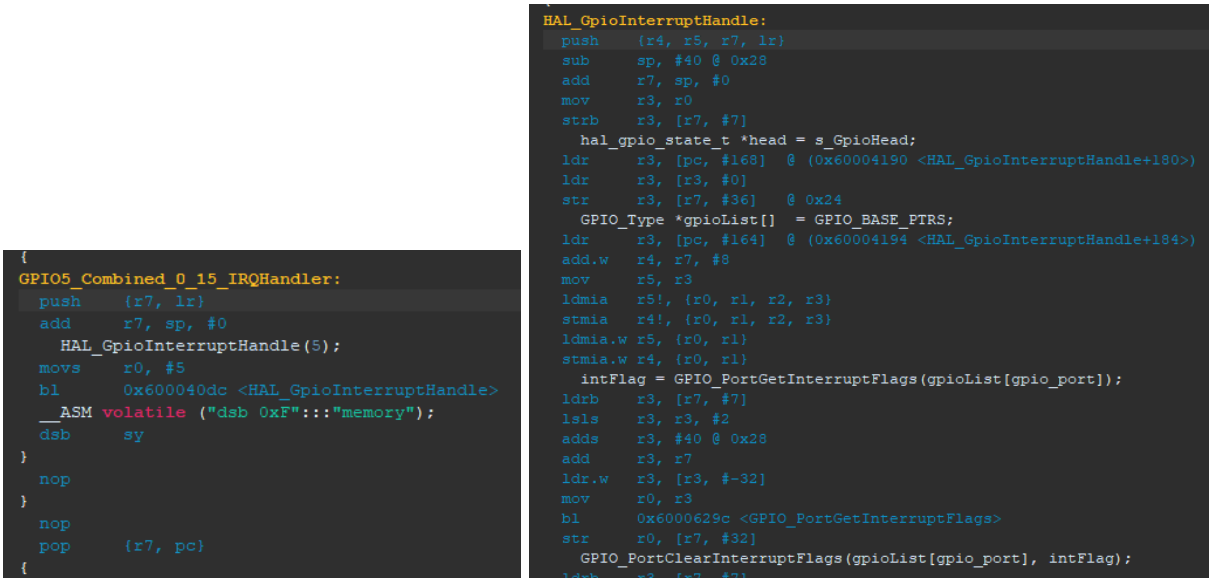

Figure 8 – Disassembly of the Callback Invocation:

The disassembly reveals multiple instructions that occur before the call to `BOARD_INITPINS_USER_BUTTON_callback()`. The complexity of the execution flow in this approach explains the increase in observed latency.

Measurement with HAL (example):

- t_1 = 405.28809 ns × 0.6 = 243 cycles
- t_2 = 52.85642 ns × 0.6 = 32 cycles
- t_3 = 2 + t_2 = 34 cycles
- td = 243 – 34 = 209 cycles

In comparison mode, the function `HAL_GpioInterruptHandle()` performs several operations to identify which pin triggered the interrupt:

- Reading flag registers
- Applying masks to check the active pin
- Searching through the list of registered callbacks
- Calling the function `BOARD_INITPINS_USER_BUTTON_callback()`

Each of these steps adds dozens of instructions, significantly increasing the response latency.

This experiment highlights the performance difference between a lean ISR and a software abstraction with additional functionalities, making it ideal for real-time analysis in embedded systems.

3.3 FreeRTOS with Semaphore

Before running the tests with FreeRTOS, it is necessary to change the active project in MCUXpresso:

Compilation:

- Switch to the `free_rtos` folder in the cloned repository.
- Import the corresponding project into MCUXpresso.
- Compile the project before flashing it to the board.

In this approach, a high-priority task is used to toggle the debug pin in response to a semaphore released within the ISR. The goal is to measure the latency associated with the context switch between the interrupt and task execution.

```
void GPT2_IRQHandler(void)
{
GPT2->SR = GPT_OC3_FLAG;
// Clear GPT output compare interrupt flag
xSemaphoreGiveFromISR(xSemaphore_consumer, &xHigherPriorityTaskWoken);
//Signal the consumer task
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
// Force context switch if necessary
__DSB(); // Data Synchronization Barrier for memory operations
}
```

Operation:

- The interrupt is triggered by the GPT2_COMPARE3 channel.
- The ISR simply releases the semaphore and requests a context switch.
- The `producer_task`, which has the highest priority, is unblocked and toggles the debug GPIO.

This configuration simulates a typical multitasking environment in embedded systems using an RTOS.

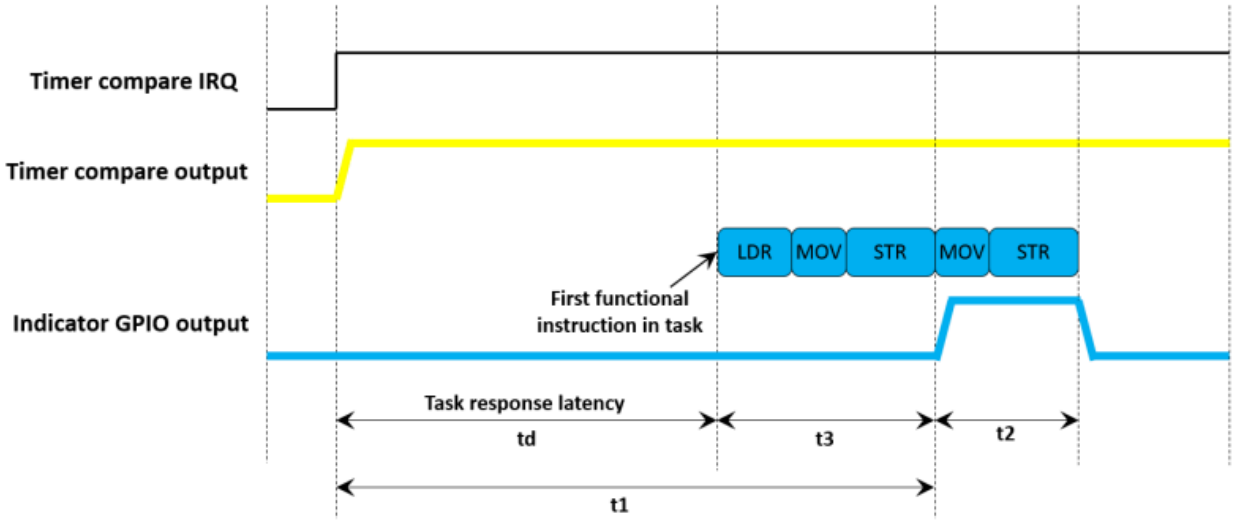

Figure 10 – Latency Measurement Diagram with FreeRTOS【1】

This image presents a schematic representation of the latency measurement process with FreeRTOS. The interrupt event is generated by the GPT2_COMPARE3【2】comparator and captured on oscilloscope channel 1. Instead of the ISR directly manipulating the GPIO, it only releases a semaphore, and a high-priority task is scheduled by the kernel to toggle GPIO1_IO19, which is observed on channel 2.

The diagram highlights the extra time introduced by the context switch and the RTOS scheduling mechanism. The distance between the two events represents the actual delay in the system's response in a multitasking environment. The diagram illustrates the difference between the interrupt event (channel 1) and the system response performed by the task (channel 2). The presence of the RTOS adds intermediate steps, such as scheduling and context switching, which significantly increase the total latency.

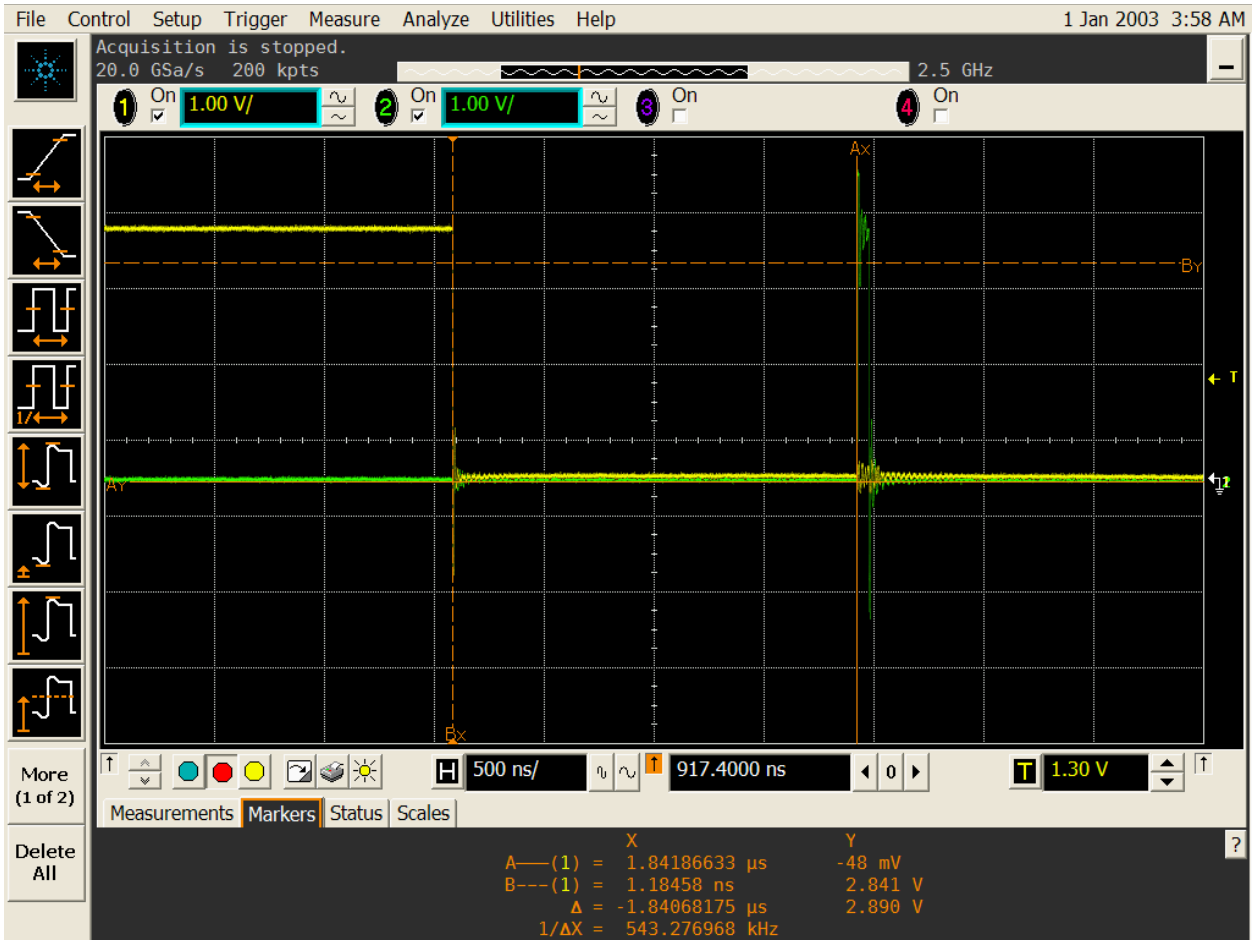

Figure 11 – Latency Measurement in FreeRTOS:

In this oscilloscope capture, a significant delay can be observed between the falling edge of the interrupt signal and the rising edge of the debug pulse. This represents the time required for the FreeRTOS kernel to perform the context switch from the interrupt to the task responsible for handling the response.

Example Measurement:

- t_1 = 1840.68175 ns × 0.6 = 1104 cycles
- t_2 = 53.55003 ns × 0.6 = 32 cycles
- t_3 = 2 + t_2 = 34 cycles
- td = 1104 – 34 = 1070 cycles

This experiment demonstrates how the addition of an RTOS, while providing structure and scalability to the software, introduces a latency penalty that must be taken into account in time-sensitive real-time applications.

---

4. Results Table

Function	t_1 (ns)	t_2 (ns)	t_3 (cycles)	td (cycles)
Bare Metal (GPT2)				
External GPIO (no compare				
External GPIO (with HAL)				
FreeRTOS + Semaphore				

---

5. Technical Challenge

During the experiments with GPT2, a hardware limitation was identified: only channel 1 of the timer can automatically reset the counter【3】. To generate accurate periodic interrupts using channels 2 or 3, it is necessary to configure channel 1 as the master, enabling synchronized system control.

Challenge Objective
Configure the GPT to generate interrupts precisely every 1.37 seconds using channel 3, with automatic reset via channel 1.

Implementation Tip: Peripheral Drivers via Config Tools

To simplify the timer configuration, it is recommended to use MCUXpresso Config Tools, which automatically generates Peripheral Drivers for initializing the GPT.

Clock and divider configuration can also be done graphically, reducing errors and saving development time.

Challenge Parameters

- Base GPT frequency: 75 MHz
- Crystal divider (pre-scaler): values between 1 and 4096
- Internal clock divider: values between 1 and 16

Steps to Follow:

1. Choose appropriate dividers to obtain a counting frequency that allows the accurate generation of 1.37 seconds:
   f_timer = 75,000,000 ÷ (divisor_cristal × clock_divider)
2. Calculate the corresponding compare value:
   Compare value = 1.37 × f_timer
3. Configure GPT1 as the master (channel 1 enabled with auto-reset mode).
4. Implement the logic using the generated Peripheral Drivers and test with an oscilloscope to validate the precision of the generated signal.

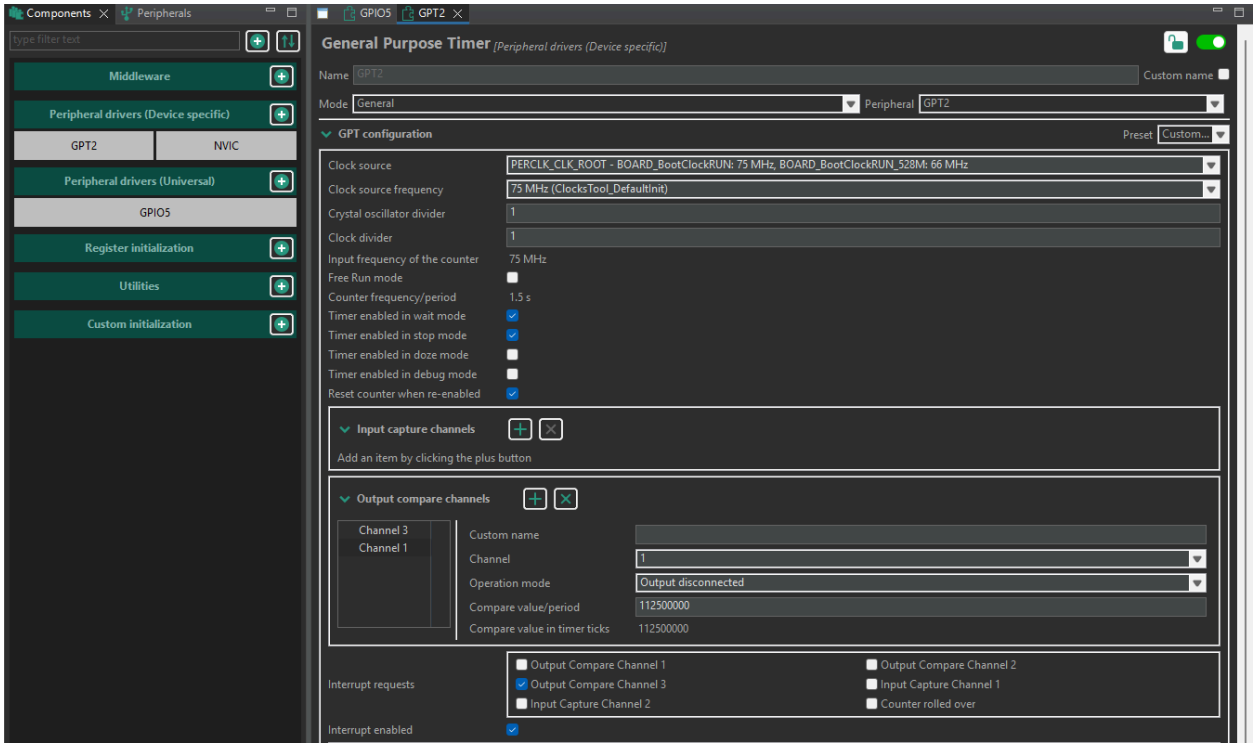

Figure 12 – Master Channel (GPT) Configuration with MCUXpresso Config Tools

Questions:

1. Does implementing a master channel (Channel 1) to reset the counter impact the interrupt response delay on Channel 3? Justify your answer.

---

2. In a system running FreeRTOS, does the interrupt latency remain constant under multiple events? Which factors can cause this delay to vary?

---

3. When using GPT Channel 1 as a master to reset the counter, what kind of overhead is introduced into the system? Does this overhead affect only the timing of the pulse generation, or does it also impact the ISR execution of the slave channel (Channel 3)?

---

4. What are the risks of relying on multiple compare channels in the GPT when using FreeRTOS with low-priority tasks executing critical operations (e.g., flash writing)?

---

5. Can the `portYIELD_FROM_ISR()` function in FreeRTOS introduce additional delay in the response to an event? In what scenarios might this become critical?

---

6. When comparing interrupt latencies between a bare-metal approach and a FreeRTOS-based approach, what do you observe in terms of jitter (latency variation)? What are the main causes of this behavior?

---

6. Reference

【1】NXP Semiconductors. AN12078 – Measuring Interrupt Latency on i.MX RT Series
https://www.nxp.com/docs/en/application-note/AN12078.pdf

【2】NXP Community. Accessibility of GPT2_COMPARE3 on the MIMXRT1050-EVKB Board, technical forum, 2022.
https://community.nxp.com/t5/i-MX-RT-Crossover-MCUs/Accessibility-of-GPT1-COMPARE1-on-the-MX-RT1050-EVKB-board/td-p/1494034

【3】NXP RM – i.MX RT1050 Processor Reference Manual, section 51.5.1.1 (Restart Mode)
https://www.nxp.com/webapp/Download?colCode=IMXRT1050RM