STM32 - Cortem-M4

The ADC (Analog-to-Digital Converter)

The ADC is a peripheral that measures an analog voltage (e.g., from a temperature sensor, microphone, or potentiometer) and converts it into a digital number that the CPU can understand and process.

Types of ADCs

While there are many ADC architectures, the most common type found in microcontrollers like the STM32 is the **Successive Approximation Register (SAR) ADC**, which offers a good balance of speed and resolution.

ADC Type	Resolution	Reference Voltage	Conversion Time	Typical Formula
Flash	High (10-12 bit)	Internal or external	Very fast (few ns)	N/A
Successive Approximation	12 bit (configurable)	Vref	~number of bits + 1 clocks	Digital Output = (Vin/Vref) * (2^Resolution - 1)
Dual Slope	High (12-16 bit)	Vref	Slow (ms)	Digital Output = (Charge/Discharge time proportional)

Fundamental ADC Characteristics

1. **Resolution:** The number of bits the ADC uses to represent the analog voltage. A higher resolution means more discrete steps and a more precise measurement.
   - An 8-bit ADC has 2^8 = 256 steps.
   - A 12-bit ADC has 2^12 = 4096 steps.
2. **Reference Voltage (Vref):** The maximum analog voltage that the ADC can measure. Any input voltage is measured as a fraction of this reference. On most STM32s, this is tied to the supply voltage (VDDA).
3. **Step Size:** The smallest change in analog voltage that the ADC can detect. It is the resolution of the measurement.

   - **Step Size = Vref / (Number of Steps)**
   - Example: With `Vref = 3.3V` and a 12-bit ADC, Step Size = `3.3V / 4096 ≈ 0.0008V` or `0.8mV`.
4. **Conversion Time:** The time it takes for the ADC to perform one full conversion. For a SAR ADC, this is roughly proportional to its resolution (e.g., a 12-bit conversion takes about 12-14 ADC clock cycles).

The Core ADC Formula

The relationship between the input voltage and the digital output is a simple ratio:

```
Digital_Output = (Input_Voltage / Vref) * (2^Resolution - 1)
```

Example:

- System: 12-bit ADC with `Vref = 3.3V`.
- Input: `Vin = 1.5V`.
- `Digital_Output = (1.5V / 3.3V) * 4095 ≈ 1861`.

Features of the STM32 ADC

- **Resolution:** 12-bit native resolution, configurable down to 10, 8, or 6 bits for faster conversions.
- **Successive Approximation Architecture:** Provides a good balance between speed and precision.
- **Multiple ADCs & Channels:** Most STM32s have multiple ADC units (ADC1, ADC2, ADC3) with many multiplexed input channels, allowing you to measure signals from dozens of different pins.
- **Flexible Conversion Modes:**
   - **Single Conversion:** Convert one channel, one time. Simple but inefficient if done by polling.
   - **Continuous (Burst) Mode:** Repeatedly convert the same channel as fast as possible.
   - **Scan Mode:** Automatically convert a sequence of pre-selected channels one after another.
   - **Discontinuous Mode:** A variation of scan mode for converting a small subset of the sequence at a time, triggered by an event.
- **ADC Clock:** The ADC has its own clock, derived from the APB2 bus, which must be configured within a specific range (e.g., 0.6 MHz to 36 MHz) for accurate conversions.

Programming the STM32 ADC (Single Conversion, Polling)

This is the simplest way to get a reading. The CPU starts a conversion, waits for it to finish, and then reads the result.

Configuration Steps:

1. **Enable Clocks:** Enable the peripheral clocks for both the GPIO port of the analog pin and the ADC itself.
2. **Configure GPIO Pin:** Set the target pin to **Analog Mode** (`MODER` bits = `11`). This disconnects the pin from the digital I/O buffers to prevent interference.
3. **Configure ADC:**
   - Set the desired resolution (e.g., 12-bit).
   - Set the conversion mode (e.g., single conversion).
   - Define the conversion sequence. For a single channel, this means setting the number of conversions to 1 and placing the channel number in the first sequence position.
4. **Enable ADC:** Turn the ADC on by setting the `ADON` bit in `ADC_CR2`.

Reading a Value:

1. **Start Conversion:** Set the `SWSTART` bit in `ADC_CR2` to begin a software-triggered conversion.
2. **Wait for Completion:** Poll the `EOC` (End of Conversion) flag in the `ADC_SR` (Status Register) until it is set by the hardware.
3. **Read Result:** Read the 16-bit `ADC_DR` (Data Register) to get the conversion result. Reading this register automatically clears the `EOC` flag.

Bare-Metal Code Example:

```
// Assumes the analog sensor is connected to PA0 (ADC1_IN0)
void ADC_Init(void) {
    // 1. Enable peripheral clocks
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // GPIOA clock
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;  // ADC1 clock

    // 2. Configure PA0 as an Analog pin
    GPIOA->MODER |= (3 << (0 * 2)); // Set bits for pin 0 to 11 (Analog mode)

    // 3. Configure ADC parameters
    ADC1->CR1 &= ~ADC_CR1_RES;   // Resolution to 12-bit (default)
    ADC1->CR2 &= ~ADC_CR2_CONT;  // Single conversion mode

    // 4. Configure the conversion sequence
    ADC1->SQR1 |= (0 << ADC_SQR1_L_Pos);   // Sequence length = (0000) 1 conversion
    ADC1->SQR3 |= (0 << ADC_SQR3_SQ1_Pos); // First (and only) conversion is on Channel 0

    // 5. Enable the ADC
    ADC1->CR2 |= ADC_CR2_ADON;
}

uint16_t ADC_GetValue(void) {
    // 1. Start the ADC conversion
    ADC1->CR2 |= ADC_CR2_SWSTART;

    // 2. Wait until the End of Conversion (EOC) flag is set
    while(!(ADC1->SR & ADC_SR_EOC));

    // 3. Read the converted value from the Data Register
    //    Reading the DR also clears the EOC flag.
    uint16_t val = ADC1->DR;
    return val;
}
```

The DAC (Digital-to-Analog Converter)

The DAC performs the reverse operation of the ADC. It takes a digital number from the CPU and converts it into a corresponding analog voltage level.

This is essential for applications that need to generate analog signals, such as:

- Driving audio amplifiers.
- Controlling the brightness of an analog-driven display.
- Generating control voltages for motors or other analog hardware.
- Creating arbitrary waveforms for testing and signal generation.

Fundamental DAC Characteristics

- **Resolution:** The number of bits in the digital input. STM32 DACs are typically 12-bit, providing 4096 distinct output voltage levels.

- **Reference Voltage (Vref):** The maximum analog voltage the DAC can produce. The output is a fraction of this voltage.
- **Output Voltage Formula:** `V_out = Vref * (Digital_Input / 2^Resolution)`
- **Settling Time:** The time it takes for the analog output to stabilize to its new value after the digital input is changed.

Features of the STM32 DAC

1. **Dual Channels:** Most STM32s include a DAC with two independent output channels (typically on pins `PA4` and `PA5`).
2. **12-Bit Resolution:** Provides fine-grained control over the output voltage.
3. **Buffered Output:** An optional internal op-amp buffer can be enabled to provide a low-impedance output capable of driving external loads without voltage drop.
4. **Flexible Triggers:** A DAC conversion (updating the output) can be triggered by:
   - **Software:** Writing to a trigger register.
   - **Hardware:** An external pin event or, most commonly, a **timer update event**.
5. **DMA Support:** For generating complex waveforms (like a sine wave), DMA can be used to stream a buffer of digital values directly to the DAC, freeing the CPU from having to manually update the output at high speed.

Programming the STM32 DAC (Software Trigger)

This is the simplest way to use the DAC: your software explicitly tells the DAC when to update its output to a new value.

Configuration Steps:

1. **Enable Clocks:** Enable the clocks for the GPIO port (e.g., GPIOA) and the DAC peripheral.
2. **Configure GPIO Pin:** Set the DAC output pin (e.g., `PA4` for DAC1) to **Analog Mode**.
3. **Configure DAC:** In the `DAC_CR` register:
   - Enable the DAC channel (`ENx`).
   - Enable the trigger (`TENx`).
   - Select the "Software Trigger" (`TSELx` = `111`).
   - (Recommended) Enable the output buffer (`BOFFx` = `0`).

Setting a Value:

1. **Write Data:** Write the 12-bit digital value you want to output to the appropriate **Data Holding Register** (e.g., `DHR12R1` for 12-bit, right-aligned data on channel 1).
2. **Trigger Conversion:** Set the corresponding bit in the **Software Trigger Register** (`SWTRIGR`). This causes the value from the DHR to be transferred to the conversion logic, updating the analog output.

Bare-Metal Code Example:

```
// Assumes controlling an LED or similar on PA4 (DAC_OUT1)
void DAC_Init(void) {
    // 1. Enable peripheral clocks
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;

    // 2. Configure PA4 in Analog Mode
    GPIOA->MODER |= (3 << (4 * 2));

    // 3. Configure DAC Channel 1
    //    Enable channel, enable trigger, select software trigger (TSEL=111)
    DAC->CR = DAC_CR_EN1 | DAC_CR_TEN1 | DAC_CR_TSEL1;
}

void DAC_SetValue(uint16_t val) {
    // 1. Write the 12-bit value to the Data Holding Register for Channel 1.
    DAC->DHR12R1 = val;

    // 2. Trigger the conversion via software.
    DAC->SWTRIGR |= DAC_SWTRIGR_SWTRIG1;
}
```

Advanced Use Case: DAC with Timer Trigger

A more powerful way to use the DAC is to have it automatically update its output at a periodic rate, driven by a timer. This is the foundation for generating waveforms.

- **DAC Configuration:** Instead of selecting a software trigger, you select a timer trigger (e.g., `TSELx` = `000` for TIM6 TRGO event).
- **Timer Configuration:**
  - Set up a basic timer (like TIM6) to generate a periodic update event at the desired sample rate (e.g., every 20ms).
  - In the timer's `CR2` register, set the `MMS` bits to `010` to make the **Update Event** its **Trigger Output (TRGO)**.
- **Operation:** Now, every time the timer overflows (ARR), its TRGO signal will automatically trigger the DAC to perform a new conversion using whatever value is currently in its Data Holding Register. You can then use the timer's *interrupt* to update the `DHR` just before the next trigger occurs, allowing you to stream a sequence of values to the DAC to create a waveform.

Brown-Out Detector (BOD)

A **Brown-Out Detector (BOD)**, also known as a Brown-Out Reset (BOR) circuit, is a critical hardware safety feature. Its sole purpose is to monitor the microcontroller's supply voltage (`Vdd`).

- **Function:** If the supply voltage drops below a pre-defined threshold, the BOD will automatically force the microcontroller into a reset state. It will hold the chip in reset until the voltage rises back above the threshold, ensuring the CPU only operates when the power supply is stable.
- **Why is it essential?** An unstable or low power supply can cause unpredictable behavior in digital logic. The CPU might execute instructions incorrectly, corrupt memory, or enter a latched-up state. The BOD prevents this by ensuring the core never runs under these unsafe conditions.
- **Causes of Brown-Out:**
  - **Power Supply Issues:** An overloaded or failing power supply.
  - **High Current Draw:** A sudden current spike (e.g., starting a motor) can temporarily pull down the supply voltage.
  - **Aging Components:** "Browning" or degradation of power supply components over time.

On STM32 microcontrollers, the BOR circuitry is tightly integrated with the Power-On Reset (POR) system and is typically enabled by default.

STM32 Reset Sources

A "reset" isn't a single event; different conditions can trigger it, and they have different effects on the system's state. Knowing the source of a reset is a powerful debugging tool.

The `RCC_CSR` (Clock Control & Status Register) contains flags that tell you what caused the last reset.

Types of Resets:

1. Power Reset (Highest Priority)

This is the most comprehensive reset, clearing almost all registers in the MCU (except for the backup domain, which holds RTC data). It occurs under three conditions:

- **Power-On Reset (POR):** When power is first applied to the device.
- **Power-Down Reset (PDR):** When power is removed.
- **Brown-Out Reset (BOR):** Triggered by the BOD when `Vdd` drops too low.

2. System Reset

This is a less drastic reset that clears most registers but preserves the `RCC_CSR` so you can diagnose the cause. A system reset is generated by one of the following:

- **External Reset:** A low level on the `NRST` pin. This is your physical reset button.
- **Window Watchdog (WWDG) Reset:** The WWDG timer expires.
- **Independent Watchdog (IWDG) Reset:** The IWDG timer expires.
- **Software Reset:** Your code deliberately triggers a reset by writing to the `SYSRESETREQ` bit in the core's control registers.
- **Low-Power Management Reset:** When exiting from certain low-power modes like Standby.

By checking the flags in `RCC_CSR` at the beginning of your `main()` function, your application can determine *why* it started, which is crucial for fault analysis.

STM32 Clock Configuration

An STM32 microcontroller is a complex system of interconnected peripherals, and they all need a clock signal to operate. The **Reset and Clock Control (RCC)** peripheral is the conductor of this orchestra, responsible for generating and distributing clocks throughout the chip.

Clock Sources

STM32 MCUs offer several clock sources, each with different characteristics:

Source	Name	Description	Use Case
Internal	**HSI** (High-Speed Internal)	A factory-calibrated 16 MHz RC oscillator.	Default clock after reset. Quick startup, but less accurate than a crystal.
External	**HSE** (High-Speed External)	Connects to an external crystal oscillator. On the STM32F407 Discovery board, this is an 8 MHz crystal.	The primary source for generating the high-speed system clock via the PLL. Stable and precise.

Source	Name	Description	Use Case
Internal	**LSI** (Low-Speed Internal)	A low-power ~32 kHz RC oscillator.	Used for the Independent Watchdog (IWDG) and as a backup for the RTC.
External	**LSE** (Low-Speed External)	Connects to a 32.768 kHz external crystal (like a watch crystal).	The preferred, most accurate source for the Real-Time Clock (RTC). (Note: Not fitted on all boards).

The Clock Tree and the PLL

Out of the box, the STM32 starts up running on the internal 16 MHz HSI. To achieve the high performance these chips are capable of (e.g., 168 MHz on the STM32F407), we must use the **Phase-Locked Loop (PLL)**.

The PLL is a frequency synthesizer. It takes a stable input clock (usually the HSE) and multiplies it up to a much higher frequency. This high-frequency clock is then divided down to provide clocks for the various buses and peripherals.

The clock configuration process involves solving a set of equations to get from your input clock to your desired system clock.

Key Clock Domains:

- `Fvco`**:** The internal high-frequency output of the PLL's Voltage-Controlled Oscillator. Must be within a specific range (e.g., 100-432 MHz).
- `Fcclk` **(or** `SYSCLK`**):** The main System Clock that feeds the CPU core.
- `Fahb` **(HCLK):** The clock for the AHB bus, which connects high-speed peripherals like memory and DMA.
- `Fapb1` **(PCLK1):** The clock for the APB1 bus (lower-speed peripherals like I2C, UARTs, and most timers).
- `Fapb2` **(PCLK2):** The clock for the APB2 bus (higher-speed peripherals like SPI1, ADC1, and advanced timers).
- `Fusb`**:** A dedicated 48 MHz clock for the USB peripheral, also derived from the PLL.

Illustration: Clock Calculation for 168 MHz

Let's walk through the calculations to get a 168 MHz system clock from an 8 MHz external crystal (HSE), as is common on the STM32F407.

- **Target** `SYSCLK` **= 168 MHz**
- **Input** `Fin` **(HSE) = 8 MHz**

1. **Calculate** `M` **(PLL Input Divider):**

   - The PLL input clock (`Fvcoin = Fin / M`) must be between 1 and 2 MHz.
   - To get `Fvcoin = 1 MHz` from our 8 MHz `Fin`, we choose **`M = 8`**.
2. **Calculate `P` (Main PLL Divider):**
   - `SYSCLK = Fvco / P`. We want `168 = Fvco / P`. `P` can be 2, 4, 6, or 8.
   - Let's choose **`P = 2`**. This means we need `Fvco = 168 * 2 = 336 MHz`. This is within the valid `Fvco` range (100-432 MHz), so it's a good choice.
3. **Calculate `N` (Main PLL Multiplier):**
   - `Fvco = Fvcoin * N`.
   - `336 MHz = 1 MHz * N`. Therefore, **`N = 336`**.
4. **Calculate `Q` (USB Clock Divider):**
   - The USB clock needs to be exactly 48 MHz. `Fusb = Fvco / Q`.
   - `48 MHz = 336 MHz / Q`. Therefore, **`Q = 7`**.
5. **Calculate Bus Prescalers:**
   - **AHB Prescaler (H):** We want the AHB bus to run at its max speed. `Fahb = SYSCLK / H`. To get `Fahb = 168 MHz`, we set **`H = 1`**.
   - **APB1 Prescaler (P1):** The max frequency for APB1 is 42 MHz. `Fapb1 = Fahb / P1`. `42 MHz = 168 MHz / P1`. Therefore, **`P1 = 4`**.
   - **APB2 Prescaler (P2):** The max frequency for APB2 is 84 MHz. `Fapb2 = Fahb / P2`. `84 MHz = 168 MHz / P2`. Therefore, **`P2 = 2`**.

We have now determined all the necessary configuration values: **M=8, N=336, P=2, Q=7, H=1, P1=4, P2=2**.

Clock Configuration Programming Sequence

Configuring the clock is one of the very first things your `SystemInit()` function does after a reset. The sequence is critical, as you cannot switch to a clock source that is not yet ready.

1. **Enable HSE and wait for it to stabilize.**
2. **Configure Flash Memory:** Before increasing the system speed, you must configure the Flash memory's wait states. The Flash is slower than the CPU, so you must tell it to wait for a certain number of clock cycles on each access. At 168 MHz, this is typically 5 wait states. You also enable the instruction and data caches and the prefetch buffer for performance.
3. **Configure Bus Prescalers:** Set the H, P1, and P2 prescalers in the `RCC_CFGR` register.
4. **Configure the Main PLL:** Write the M, N, P, and Q values to the `RCC_PLLCFGR` register and select HSE as the PLL source.
5. **Enable the PLL and wait for it to lock.**
6. **Switch System Clock to PLL:** Change the `SW` bits in `RCC_CFGR` to select the PLL as the system clock source.
7. **Wait for the Switch to Complete:** Poll the `SWS` bits to confirm that the system is now running from the PLL.

8. **Update SystemCoreClock Variable:** Update the global `SystemCoreClock` variable so that other parts of the system (like the SysTick configuration) know the new clock speed.

Bare-Metal Code Example (Clock Setup)

This code implements the sequence described above to switch the system to a high-speed clock derived from the PLL.

```
// Define the PLL configuration values calculated earlier
#define PLL_M 8
#define PLL_N 336
#define PLL_P 2 // Note: The register takes ((P/2)-1)
#define PLL_Q 7

// Define bus prescalers
#define PRE_H  1 // AHB
#define PRE_P1 4 // APB1
#define PRE_P2 2 // APB2

void Clock_Setup(void) {
    // 1. Enable HSE and wait for it to become ready
    RCC->CR |= RCC_CR_HSEON;
    while(!(RCC->CR & RCC_CR_HSERDY));

    // Enable the power interface clock, required for voltage scaling
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    // Set voltage scaling to Scale 1 for operation up to 168 MHz
    PWR->CR |= PWR_CR_VOS;

    // 2. Configure AHB and APB bus prescalers
    // AHB Prescaler = /1
    RCC->CFGR &= ~RCC_CFGR_HPRE;
    // APB1 Prescaler = /4
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV4;
    // APB2 Prescaler = /2
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;
    // 3. Configure the main PLL
    RCC->PLLCFGR = (PLL_M << RCC_PLLCFGR_PLLM_Pos)
                 | (PLL_N << RCC_PLLCFGR_PLLN_Pos)
                 | (((PLL_P / 2) - 1) << RCC_PLLCFGR_PLLP_Pos)
                 | (PLL_Q << RCC_PLLCFGR_PLLQ_Pos)
                 | RCC_PLLCFGR_PLLSRC_HSE; // Set HSE as PLL source

    // 4. Enable the main PLL and wait for it to lock
    RCC->CR |= RCC_CR_PLLON;
    while(!(RCC->CR & RCC_CR_PLLRDY));

    // 5. Configure Flash memory for high-speed operation
    // Enable prefetch, instruction & data caches, and set 5 wait states for 168MHz
    FLASH->ACR = FLASH_ACR_PRFTEN | FLASH_ACR_ICEN | FLASH_ACR_DCEN | FLASH_ACR_LATENCY_5WS;

    // 6. Select the main PLL as the system clock source
    RCC->CFGR &= ~RCC_CFGR_SW;      // Clear SW bits
    RCC->CFGR |= RCC_CFGR_SW_PLL;   // Set SW to 10 for PLL

    // 7. Wait until the PLL is being used as the system clock
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);

    // 8. Update the CMSIS SystemCoreClock variable with the new clock speed
    SystemCoreClockUpdate();
}
```

STM32 DMA Controller

So far, we have configured and used peripherals like the ADC and DAC by having the CPU directly read or write to their data registers. This works, but it's often incredibly inefficient. Now let's discuss how to offload the repetitive, high-volume work of data transfer from the CPU to a specialized co-processor: the **DMA Controller**.

Part 1: The Problem - Why the CPU is a Bottleneck

Imagine you need to sample an audio signal from an ADC at 48,000 times per second. Let's look at how we would do this *without* DMA.

- **Polling Method:** The CPU would get stuck in a loop, constantly checking the ADC's "End of Conversion" flag, reading the value, storing it in memory, and starting the next conversion. The CPU would be **100% occupied** and unable to do any other work.
- **Interrupt Method:** This is better. The CPU can do other work, but 48,000 times per second, it will be interrupted. Each interrupt forces a context switch, runs the ISR (to read the ADC value and store it), and then performs a context restore. For high-speed data streams, the **overhead of handling these interrupts can consume a significant portion of the CPU's processing time**, leaving little room for your main application logic.

In both cases, the **CPU is acting as a very expensive data courier**, simply moving bytes from a peripheral to memory. This is a waste of a powerful processing core.

Part 2: The Solution - Direct Memory Access (DMA)

The **Direct Memory Access (DMA) Controller** is a dedicated, intelligent hardware peripheral that acts as a high-speed, independent data courier. Its sole purpose is to move data between peripherals and memory, or between two memory locations, **without any CPU intervention**.

> **The Office Analogy:** Think of the **CPU** as a highly-paid, expert manager. Think of **DMA** as a silent, efficient office assistant.
>
> - **Without DMA:** The manager (CPU) has to stop his important work, walk over to the "in-tray" (ADC data register), pick up a document (the data), walk it over to the filing cabinet (RAM), and then walk back to his desk to resume his real work. This is a terrible use of his time.
> - **With DMA:** The manager (CPU) gives a single instruction to the assistant (DMA): "*Please take the next 1000 documents that arrive in the in-tray and file them sequentially in this cabinet drawer. Let me know when you're done.*" The manager can now focus entirely on his expert tasks, while the assistant works in parallel, handling the repetitive data movement in the background. The entire system is vastly more efficient.

The Benefits of DMA:

- **CPU Freedom:** The CPU is completely free to execute complex application code while data transfers happen in the background.
- **High Throughput:** The DMA controller has a dedicated path to the memory and peripheral bus, allowing for much higher data transfer rates than the CPU could manage through interrupts.
- **Power Efficiency:** By allowing the CPU to remain in a sleep or low-power state while waiting for a large data transfer to complete, DMA significantly reduces overall system power consumption.

Part 3: The STM32 DMA Architecture

The DMA controller in STM32F4 series MCUs is particularly powerful. It features a dual-bus architecture that allows it to access peripherals and memory concurrently.

Here is the hierarchy you need to understand:

- **DMA Controllers:** Most STM32F4 devices have two independent DMA controllers (**DMA1** and **DMA2**). This allows two high-speed data transfers to happen simultaneously without conflicting.
- **Streams:** Each DMA controller has multiple **Streams** (e.g., 8 streams per controller). A Stream is like a single, physical data-moving engine. Each stream can be configured for one transfer at a time.
- **Channels:** Each Stream can be connected to several different peripheral "request" lines, called **Channels**. A multiplexer allows you to select which peripheral's request will trigger the stream to perform a transfer.

> **Hierarchy Analogy:** Think of **DMA1** and **DMA2** as two separate office buildings. Each building has 8 **Streams** (assistants). Each assistant can be assigned to listen to requests from one of several **Channels** (different department in-trays, like UART, SPI, or ADC).

- **Arbiter:** Since multiple streams might want to access the memory bus at the same time, an **Arbiter** inside the DMA controller manages priorities. You can configure each stream with one of four priority levels (Low, Medium, High, Very High) to ensure that critical transfers always happen first.
- **FIFO Buffer:** Each stream has a small First-In-First-Out (FIFO) buffer. This allows the DMA to temporarily store a few data words, helping to smooth out timing differences between the peripheral and the memory bus, a feature known as "burst transfers".

Part 4: Configuring a DMA Transfer

Configuring a DMA transfer involves telling the "office assistant" five key pieces of information: **Where from? Where to? How much? How big? and How to behave?**

This is all done by writing to the registers for a specific DMA stream (e.g., `DMA2_Stream0`).

1. **Set the Peripheral Address (`DMA_SxPAR`):** This is the **source** or **destination** address. This is a fixed address that points to the peripheral's data register (e.g., `&ADC1->DR` or `&USART2->DR`).
2. **Set the Memory Address (`DMA_SxM0AR`):** This is the other end of the transfer. This is the address of your buffer in RAM.
3. **Set the Number of Data Items (`DMA_SxNDTR`):** This register holds the total number of data items to be transferred (e.g., 1024). This is a down-counter; the DMA decrements it after each transfer and stops when it reaches zero.
4. **Configure the Stream Control Register (`DMA_SxCR`):** This is the main configuration register where you define the behavior of the transfer. Key settings include:

- **Channel Selection (`CHSEL`):** Which peripheral channel is this stream listening to?
- **Transfer Direction (`DIR`):**
  - `00`: Peripheral-to-Memory (e.g., ADC to RAM)
  - `01`: Memory-to-Peripheral (e.g., RAM to UART)
  - `10`: Memory-to-Memory
- **Increment Mode (`MINC`, `PINC`):** Do you want to increment the memory address (`MINC`) after each transfer? (Almost always yes). Do you want to increment the peripheral address (`PINC`)? (Almost always no, since the peripheral data register is at a fixed address).
- **Data Size (`MSIZE`, `PSIZE`):** Are you moving bytes, half-words (16-bit), or full words (32-bit)? These must match the size of the peripheral's data register.
- **Circular Mode (`CIRC`):** If enabled, the DMA will automatically wrap back to the beginning of the memory buffer after it finishes, allowing for continuous data collection. Essential for things like audio sampling.
- **Interrupt Enables (`TCIE`, `HTIE`):** Do you want an interrupt when the transfer is complete (`TCIE`) or half-complete (`HTIE`)?
- **Priority Level (`PL`):** Set the stream's priority.

5. **Enable the Stream:** Finally, set the `EN` bit in the `DMA_SxCR` register. The stream is now armed and will begin transferring data as soon as it receives a request from the selected peripheral.

Part 5: Bare-Metal DMA Example Code

This code configures DMA2 Stream 0 for a Memory-to-Memory transfer, a great way to test a DMA setup.

`dma.h`

```
#ifndef DMA_H_
#define DMA_H_

#include "stm32f4xx.h"

void DMA_Init(void);
void DMA_Mem_Transfer(uint8_t *dma_mem_dst, uint8_t *dma_mem_src, uint16_t size);
extern volatile int dma2_flag;

#endif /* DMA_H_ */
```

dma.c

```
void DMA_Init(void) {
    // 1. Enable the clock for the DMA2 controller in the RCC peripheral.
    RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;

    // 2. Configure and enable the DMA stream's interrupt in the NVIC.
    // This allows us to get a callback when the transfer is complete.
    NVIC_EnableIRQ(DMA2_Stream0_IRQn);
}
```

```
void DMA_Mem_Transfer(uint8_t *dma_mem_dst, uint8_t *dma_mem_src, uint16_t size) {
    // clear DMA Stream configuration register
    // single transfer, M0 is target, single buffer mode, circular buffer disabled
    // Make sure the stream is disabled before configuring it.
    DMA2_Stream0->CR = 0x0000;
    while(DMA2_Stream0->CR & DMA_SxCR_EN); // Wait until it's truly disabled

    // Set the source memory address
    DMA2_Stream0->M0AR = (uint32_t)dma_mem_src;

    // Set the destination memory address
    DMA2_Stream0->PAR = (uint32_t)dma_mem_dst;
    // In M2M mode, PAR is used as the destination

    // Set the number of data items to transfer
    DMA2_Stream0->NDTR = size;

    // Configure the Control Register (CR) for the stream
    DMA2_Stream0->CR |= (0 << DMA_SxCR_CHSEL_Pos); // Channel 0 for M2M
    DMA2_Stream0->CR |= DMA_SxCR_MINC;            // Increment Memory address
    DMA2_Stream0->CR |= DMA_SxCR_PINC;            // Increment "Peripheral" (destination memory) address
    DMA2_Stream0->CR |= (0 << DMA_SxCR_MSIZE_Pos);  // Memory data size: byte
    DMA2_Stream0->CR |= (0 << DMA_SxCR_PSIZE_Pos);  // Peripheral data size: byte
    DMA2_Stream0->CR |= DMA_SxCR_PL_1;            // Priority: High
    DMA2_Stream0->CR |= DMA_SxCR_TCIE;            // Enable Transfer Complete Interrupt

    // Enable the stream to arm it for the transfer.
    // Since this is M2M, the transfer starts immediately.
    DMA2_Stream0->CR |= DMA_SxCR_EN;
}
```

```
// The Interrupt Service Routine for DMA2 Stream 0
volatile int dma2_flag = 0;
void DMA2_Stream0_IRQHandler(void) {
    // Check if the Transfer Complete Interrupt Flag is set
    if(DMA2->LISR & DMA_LISR_TCIF0) {
        // Clear the flag to prevent re-entering the ISR
        DMA2->LIFCR |= DMA_LIFCR_CTCIF0;
        // flag the interrupt
        dma2_flag = 1;
    }
}
```

Part 6: Interactive Q&A

1. You need to continuously sample an ADC into a 512-sample buffer. Once the buffer is full, the DMA should automatically start filling it from the beginning again while your main code processes the full buffer. Which specific DMA stream setting is essential for this?
2. In the code above, the `MINC` and `PINC` bits are both enabled. Why is this correct for a Memory-to-Memory transfer but would be wrong for an ADC-to-Memory transfer?
3. What is the primary role of the `NVIC_EnableIRQ(DMA2_Stream0_IRQn)` line in `DMA_Init()`? What would happen if you commented out this line?

Answers:

1. **Circular Mode (`CIRC`)**. Enabling Circular Mode tells the DMA controller that once the `NDTR` counter reaches zero, it should automatically reload it with the original value and reset the memory pointers back to the start of the buffer. This allows for continuous, gapless data acquisition.
2. In a Memory-to-Memory transfer, both the source and destination are buffers in RAM, so we need to increment our pointer for both after each copy. In an ADC-to-Memory transfer, the source is the ADC's Data Register (`ADCx->DR`), which is at a single, fixed address. Therefore, you would enable `MINC` (to fill your memory buffer) but **disable** `PINC` to ensure the DMA reads from the same peripheral register every time.
3. This line **enables the DMA stream's interrupt in the NVIC**. The `TCIE` bit in the DMA stream's control register only tells the stream to *generate* an interrupt signal. The `NVIC_EnableIRQ` line is what tells the CPU core to actually *listen* for that signal. If you commented it out, the DMA transfer would still complete successfully, but your `DMA2_Stream0_IRQHandler()` function would **never be called**, and your program (and CPU) would have no way of knowing when the transfer was finished.

---

perplexity

Interrupts vs. Events

What is an Interrupt?

An interrupt is a hardware signal that demands **CPU intervention**. It is a request for the processor to stop what it's doing and execute a specific piece of software.

- **The Flow:**

1. A peripheral generates an interrupt signal.
2. The signal goes to the **NVIC (Nested Vectored Interrupt Controller)**.
3. The NVIC prioritizes the request and, if enabled, forwards it to the CPU core.
4. The CPU **pauses** its current task, automatically **stacks** its context (registers), and **jumps** to a specific software function called an **Interrupt Service Routine (ISR)**.
5. The ISR code runs, services the peripheral, and then returns.
6. The CPU **unstacks** the original context and resumes the paused task.

- **Key Characteristic:** An interrupt always results in the execution of a software handler. It has an entry in the vector table. It is used when a software decision or action is required in response to a hardware signal.

- **Low Power Interaction:** The `WFI` (Wait For Interrupt) instruction puts the CPU into a low-power sleep state. The CPU will only wake up and resume execution when an interrupt occurs (which will then be serviced by its ISR).

What is an Event?

An event is a hardware signal that is used to trigger **other hardware peripherals directly**, without necessarily involving the CPU. It is a notification that something happened, but it does not demand that software be executed.

- **The Flow:**

1. A peripheral generates an event signal.
2. This signal is routed through the **EXTI (Extended Interrupts and Events Controller)**.
3. The EXTI can route this event signal directly to other peripherals (like a Timer or ADC) to trigger an action.

- **Example: ADC Triggered by a Timer** You can configure a timer to generate an "update event" every 1 millisecond. You can then configure the ADC to use this timer's event as its "start conversion" trigger. In this setup, the timer automatically triggers the ADC every millisecond. The CPU is completely uninvolved; no ISRs are run, and no clock cycles are wasted. The data transfer can then be handled by the DMA, creating a fully autonomous, CPU-free data acquisition pipeline.
- **Key Characteristic:** An event is a hardware-to-hardware signal. It does not have a vector in the NVIC and does not cause an ISR to run. It is used for creating complex, synchronized chains of hardware actions.
- **Low Power Interaction:** The `WFE` (Wait For Event) instruction puts the CPU into a low-power sleep state. The CPU will wake up when an event occurs, but unlike `WFI`, it will simply resume execution from the instruction after the `WFE`. It does not jump to an ISR. This is useful for synchronizing with hardware actions without the overhead of an interrupt.

Aspect	Interrupt	Event
Purpose	To request CPU/software intervention.	To trigger other hardware peripherals.
CPU Action	CPU stops, saves context, jumps to ISR.	CPU is typically not involved.
Software	**Requires** an Interrupt Service Routine (ISR).	**No** software handler is executed.
Controller	Managed by the **NVIC**.	Managed by the **EXTI** and peripheral logic.
Low Power	Wakes CPU from `WFI` and executes ISR.	Wakes CPU from `WFE` and continues execution.

Cortex-M Advanced Debugging

One of the most powerful features of the ARM Cortex-M architecture is its deeply integrated and sophisticated debug infrastructure. This goes far beyond simple breakpoints and provides incredible insight into your system's real-time behavior.

Serial Wire Debug (SWD): The Physical Interface

The **SWD** protocol is the modern, two-wire physical interface used to connect a debugger probe (like an ST-Link) to the ARM core's debug access port. It is the successor to the older, more pin-intensive JTAG standard.

- **SWDIO:** A single, bidirectional data pin.
- **SWCLK:** A clock signal driven by the debugger.
- **(Optional) SWO:** The **Serial Wire Output** pin, used for streaming trace data out of the chip.

Through this simple interface, a debugger can:

- Halt, run, and step the CPU.
- Read and write to any memory location or core register.
- Set hardware breakpoints.
- Program the internal Flash memory.

Instrumentation Trace Macrocell (ITM): `printf()` Style Debugging

The **ITM** is a hardware block inside the Cortex-M core designed to provide low-overhead, application-driven tracing. It is the hardware that enables "printf-style debugging" without the performance penalty of a real UART.

- **How it Works:** The core has dedicated registers that act as "stimulus ports." When your software writes a value to one of these ports, the ITM hardware packetizes that data and streams it out of the chip, typically over the **SWO** pin.
- **The `printf()` Connection:** Your IDE and debugger can redirect the standard `printf` function to write its output to an ITM stimulus port. The debugger probe then captures this SWO data stream and displays it in a console window on your PC.
- **Advantages:**
  - **Non-intrusive:** It is vastly faster and less intrusive than halting the CPU or using a slow UART.
  - **Real-time:** You can view real-time variable values and event logs from your running application without significantly affecting its timing.

Embedded Trace Macrocell (ETM): The Flight Recorder

The **ETM** is an incredibly powerful, but more complex, trace unit. While the ITM shows you what your software *tells* it to show, the ETM shows you what the CPU is *actually doing*.

- **Instruction Trace:** The ETM monitors the processor's program counter and execution flow. It generates highly compressed trace packets that describe the path of execution (e.g., "executed this branch," "entered this loop 10 times"). The debugger on the PC can then reconstruct the exact sequence of instructions that were executed leading up to an event or a crash. This is an invaluable tool for debugging complex, intermittent bugs.
- **Data Trace:** On some cores, the ETM can also trace data accesses, showing you the addresses and values of memory reads and writes.

Using ETM requires a more advanced debug probe that can capture the high-speed parallel trace data, but it provides unparalleled visibility into the core's operation.

Trace Port Interface Unit (TPIU): The Data Funnel

The **TPIU** is the final piece of the puzzle. It is the hardware block that acts as a funnel, gathering all the different on-chip trace sources (ITM, ETM, etc.) and multiplexing them onto the physical trace pins that leave the chip.

- In **Serial Wire Viewer (SWV)** mode, the TPIU serializes the ITM data and sends it out over the single `TRACESWO` pin.
- In parallel trace mode (for ETM), it drives the multi-bit `TRACEDATA[3:0]` bus and the `TRACECLK` signal.