Power – Interrupt-driven Design

- Avoid polling.
- Want to enter idle loop in OS which can put processor to sleep.

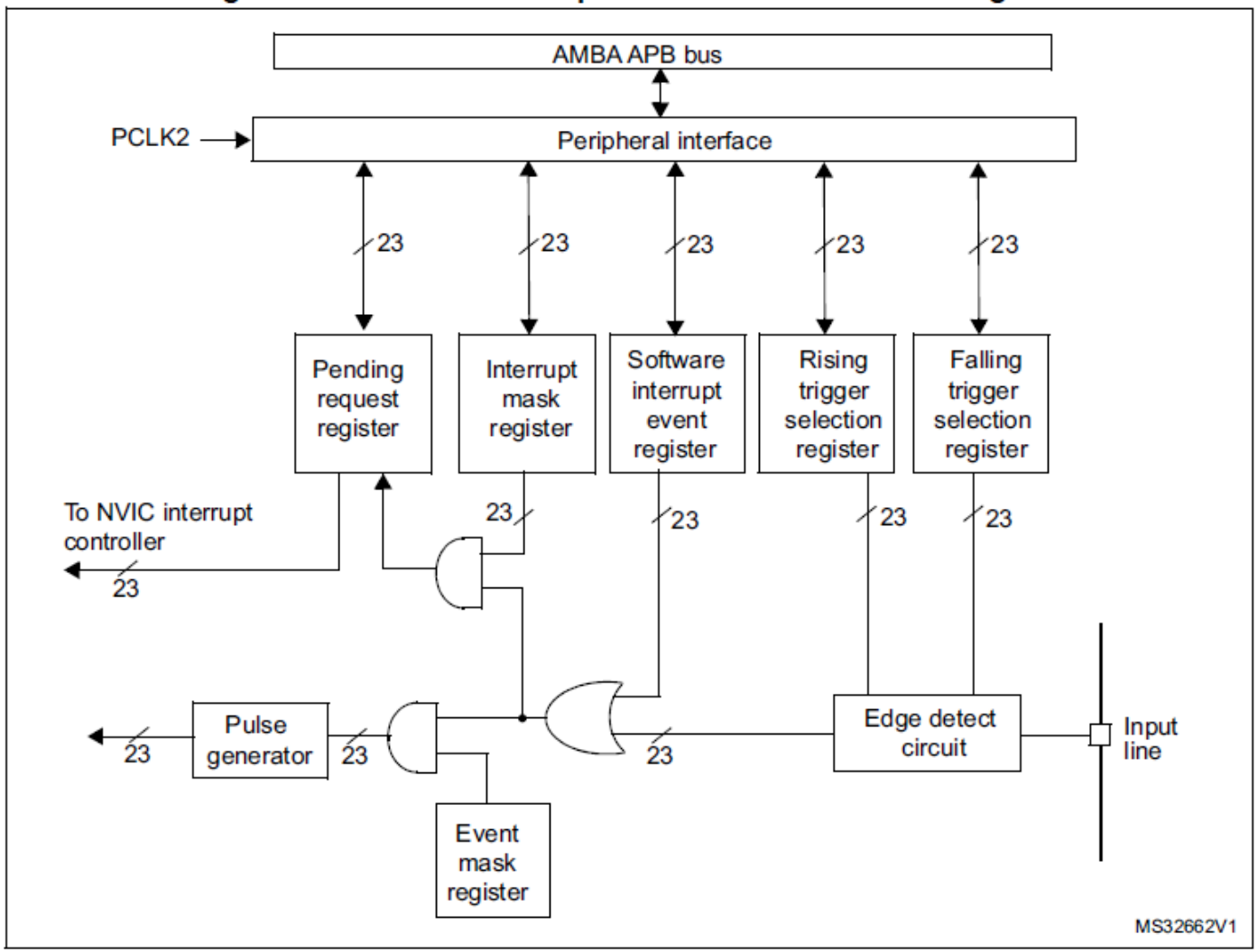
```
void QXK::onIdle(void) {
    //__WFI();   Wait-For-Interrupt
}
int_t QF::run(void) {
    QF_INT_DISABLE();
    initial_events(); // process all events posted during initialization
    onStartup(); // application-specific startup callback
    QF_INT_ENABLE();
    // the QXK idle loop...
    for (;;) {
        QXK::onIdle(); // application-specific QXK idle callback
    }
    return static_cast<int_t>(0);
}
```

Power – Interrupt-driven Design

- Key points for interrupt driven design:
  - Minimum time spent in ISR.
  - Defer processing to application.
  - Need to disable interrupt before app process it to avoid queue overflow.
  - See button example.

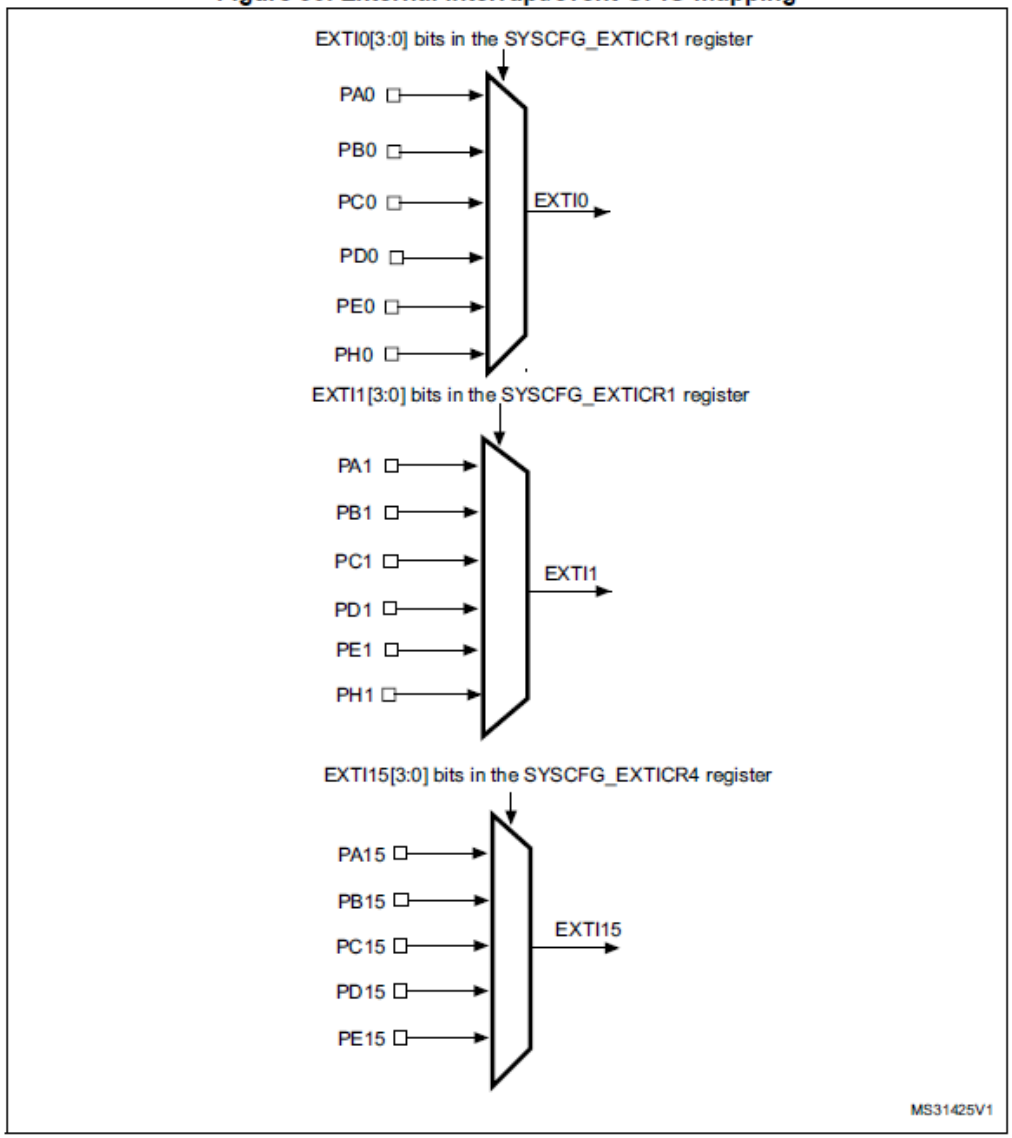
Power – Interrupt-driven Design

Figure 29. External interrupt/event controller block diagram



Power – Interrupt-driven Design

Figure 30. External interrupt/event GPIO mapping



Speed – Zero-copy

- Zero-copy
- Avoid copying data from one buffer to another along its path.
- For example, consider a hypothetical case when System AO sends data to UART:
  - ✗ System copies *data* to a buffer in an event object.
  - ✗ It then copies the event object to the event queue of UartOut.
  - ✗ UartOut copies the event object from the event queue to a local variable.
  - ✗ It then copies each *data byte* from the event buffer to the HAL UART driver through function parameter (register or stack).
  - ✗ The HAL UART driver copies each data byte to the hardware data register.

Speed – Zero-copy

- An alternate design with zero-copy:
  - First QP (or UCOS-II) moves events to and from an event queue via *pointers*.
    - It is much more efficient to just copy a 4-byte pointer than an entire event object.
    - QP has to manage event buffer allocation and garbage collection.
    - QP does it via block-based memory pools. See qmpool.h, qf_mem.cpp and qf_dyn.cpp.
    - Block-based pool is efficient since there is no need to search. There is no external fragmentation, but there is internal fragmentation.
    - Garbage collection is done via reference counters. Consider multiple subscribers. Note – Only the *event pointer* is copied multiple times to each event queue, but not the event object itself.

Speed – Zero-copy

- ✔ Secondly, the application design should *separate data and control paths*.
  - Avoid passing large amount of data via events. There is an inherit copying into and out of an event buffer. Different event buffers are not contiguous.
  - Use events mainly for control (behaviors).
  - Use a pipe (or FIFO) to pass data directly between objects. Later we will see DMA. Once data have been put into a FIFO, DMA will move them out directly from FIFO to peripheral bus. Note – contiguous memory (except wrap around).
  - On the RX side, DMA will move data directly from peripheral bus to FIFO. An application object will get it out of FIFO.

Speed – Zero-copy

- When using FIFO for direct data transfer, what are the roles of statecharts and events then?
- Statecharts and events are still essential for managing the control path.
- Firstly, FIFO is a shared resource accessed by multiple objects (threads) and possibly an ISR.

  The owner of the FIFO passes the pointer to the FIFO (via events) to other objects that need access to it.

  By following the REQ/CFM paradigm (start/stop), access to the shared resource is properly controlled. (No access to it before it is ready, or after it is freed.)
- Secondly, events are used to notify when data become available (e.g. send request) or when queue becomes empty (e.g TX flow control).
- See UartAct example.

Speed – Bulk Transfer

- The second optimization technique for speed is to use bulk transfer as much as possible.
- A small number of large transfer is more efficient than a larger number of small transfer.
- We are familiar with this: Reading multiple bytes in a single fread() is more efficient han reading one byte at a time.
- Similarly, when reading data from the ST IMU (LSM6DS0.pdf), it is much more efficient to read all available samples in a single I2C or SPI transfer.

Speed – Bulk Transfer

- See Section 3.3 Multiple reads (burst) (page 20) of LSM6DS0.pdf.
- When using high ODR, enable FIFO continuous mode.
- Note how the ST hardware automatically wrap around the register address to enable reading out multiple sets of gyro XYZ and accel XYZ in a single transfer.
- Recall overhead of I2C read:

  device address, register address write, data read…

- For single byte read, 3X of data is transferred.
- SPI is better at the expense of additional wires.

Speed - DMA

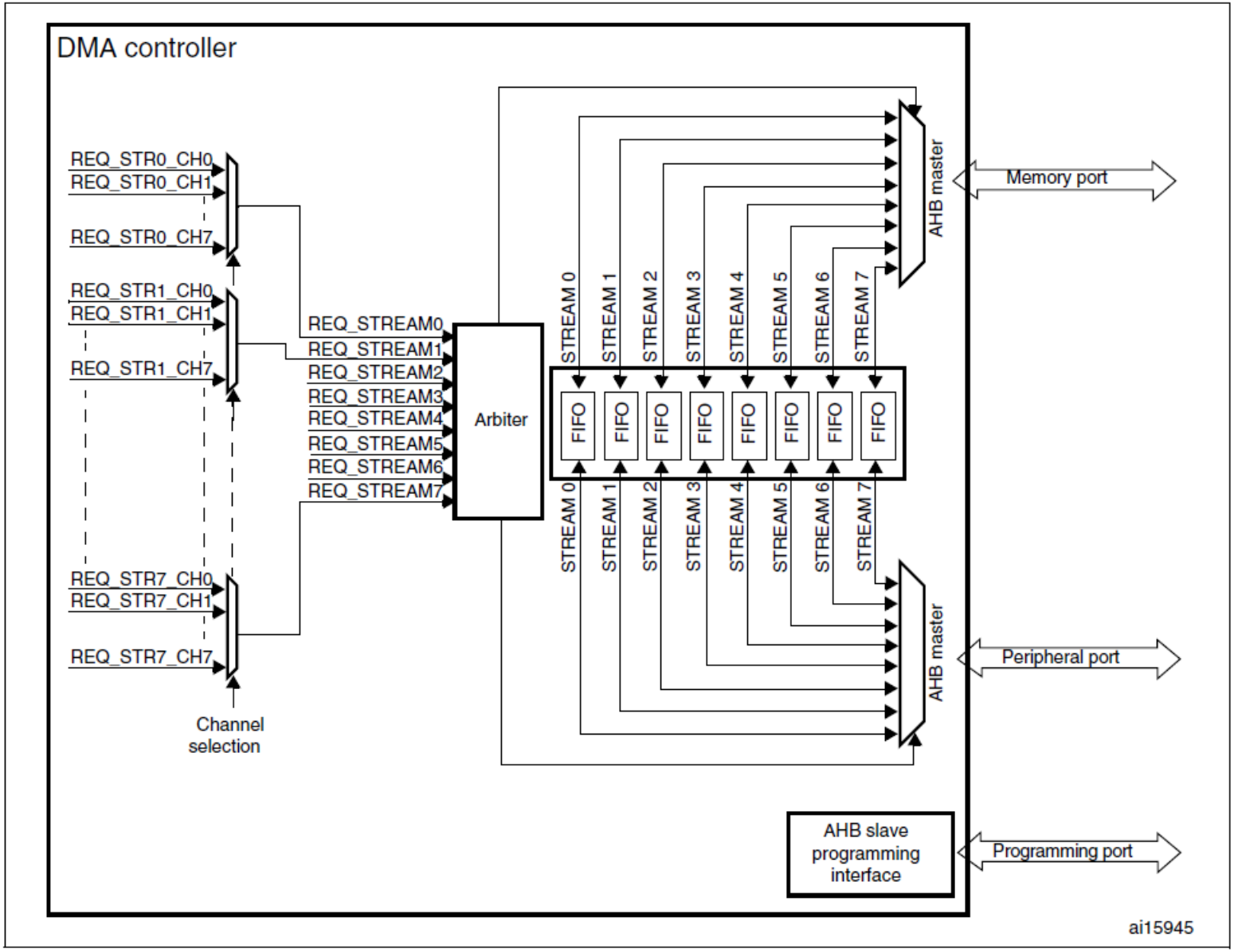
- DMA is very efficient in transferring a large amount of data between memory and peripheral (via data register).
- It can work with I2C, SPI, UART or GPIO.
- It can work in both directions (to and from peripherals).
- There is an overhead to set up each transfer, so it may not be efficient to send a small amount of data.
- It can be triggered by timer interrupt to achieve precise timing when writing to GPIO.
  - For example, we can use DMA to trigger writing to a GPIO port on every risinig/falling edge of a PWM timer.
  - PWM is used a clock signal and the GPIO port outputs data synchronously to the peripheral. This is how the LED matrix panel works.

Speed - DMA

- For UART (or SPI or I2C), when transmitting to peripheral, DMA is triggered when the data register is available for writing.
  - This is much more efficient than triggering an interrupt each time when the transmit buffer is empty.
  - Note that STM32F4 does not have data FIFO for UART, which makes DMA essential. If a deep FIFO is available, one may get by without DMA.
- When receiving data from peripheral, DMA is triggered when the receive buffer is not empty.
  - Again it is much more efficient than triggering an interrupt to copy each byte in ISR.
  - In our demo project, we only detect single character so DMA is not used.
  - A tricky point. When setting up RX DMA, we need to tell HW how many bytes are to be received. For UART, is there a problem? If so how to solve?

Speed - DMA

Figure 22. DMA block diagram



Speed - DMA

- See STM32F401 Reference Manual Section 9.3.3 for DMA channel/stream selection.

Table 28. DMA1 request mapping (STM32F401xB/C and STM32F401xD/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

Speed - DMA

- See demo project UartOut for UART TX DMA. Key points:
  - DMA transfers data out of FIFO directly. No extra copy.
  - STM32Cube driver (stm32f4xx_hal_uart.c) is a good place to start, but may not be optimized.
    - The original driver always enables the DMA half complete interrupt which we don't care here. (We would need it for the RX path. Why?)
    - It also triggers the UART TX Complete interrupt upon DMA complete interrupt. (To share common code.)
    - The result is for every DMA TX completion, 3 interrupts are generated.

Speed – Other techniques

- RTOS is a good source to get ideas since they are highly optimized.
- How does QP find the highest priority ready task to run?
  - On Cortex-M3 or above, use CLZ instruction (count leading zeros) for fast LOG2 (see qpset.h or qf_port.h):

    ```
    #define QF_LOG2(n_) ((uint_fast8_t)(32U - __CLZ(n_)))
    ```

  - Otherwise, use table-lookup (see QF_log2Lkup[ ] in qf_act.cpp). (Tradeoff between speed and code size).
- In fw_pipe.h, why does it force size to order of 2.
- Check out handy macros in fw_macro.h