EMBSYS 110, Spring 2017 Week 7 & 8

- Week 7
  - Debugging, profiling and testing.
  - Interrupt-driven design (vs polling).
  - Optimization (zero-copy, DMA).
  - Separation of control and data paths.
- Week 8
  - Cache
  - Non-blocking vs blocking.
  - Others.

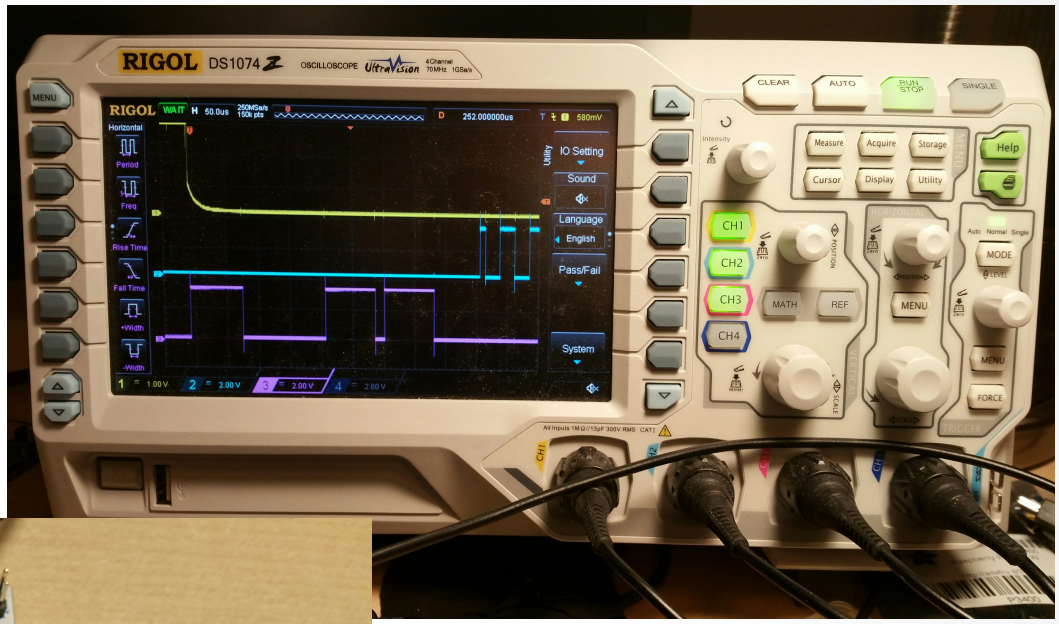
Debugging, Profiling and Testing

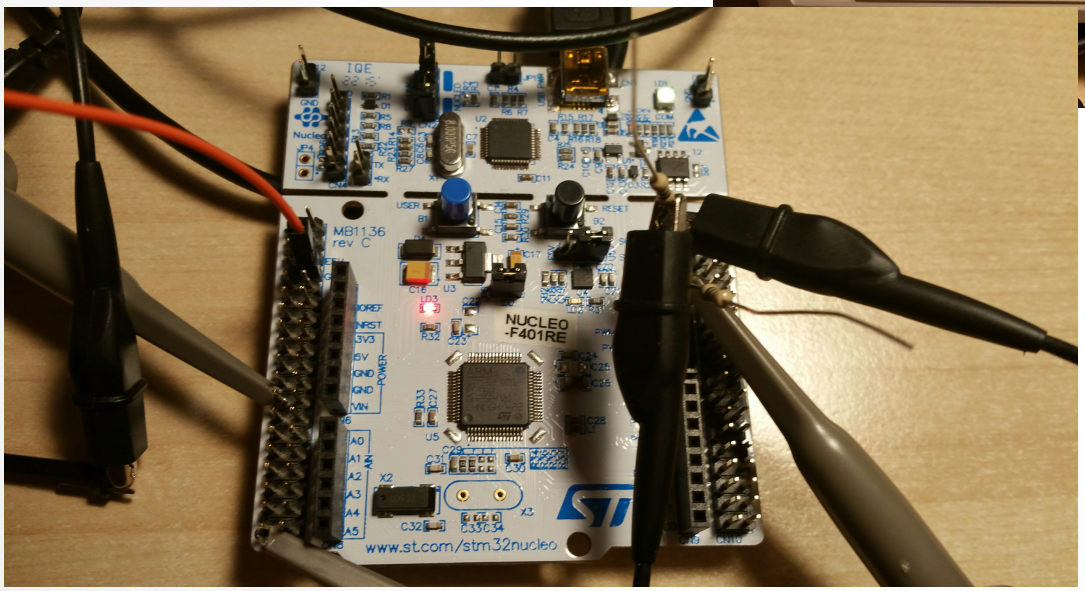
- Debugging with interactive debug console which is itself an HSM (advantages?).
- Debug level run-time setting.
- Debug on/off run-time setting per HSM.
- Debug console can show the current states of all HSM's. Global state.
- Debug console can start and stop individual HSM's. Useful for testing and diagnosis.
- It's not trivial to be able to repeatedly start and stop a module (HSM). Compare with a free-running RTOS thread. A good test for leakage.

Debugging, Profiling and Testing

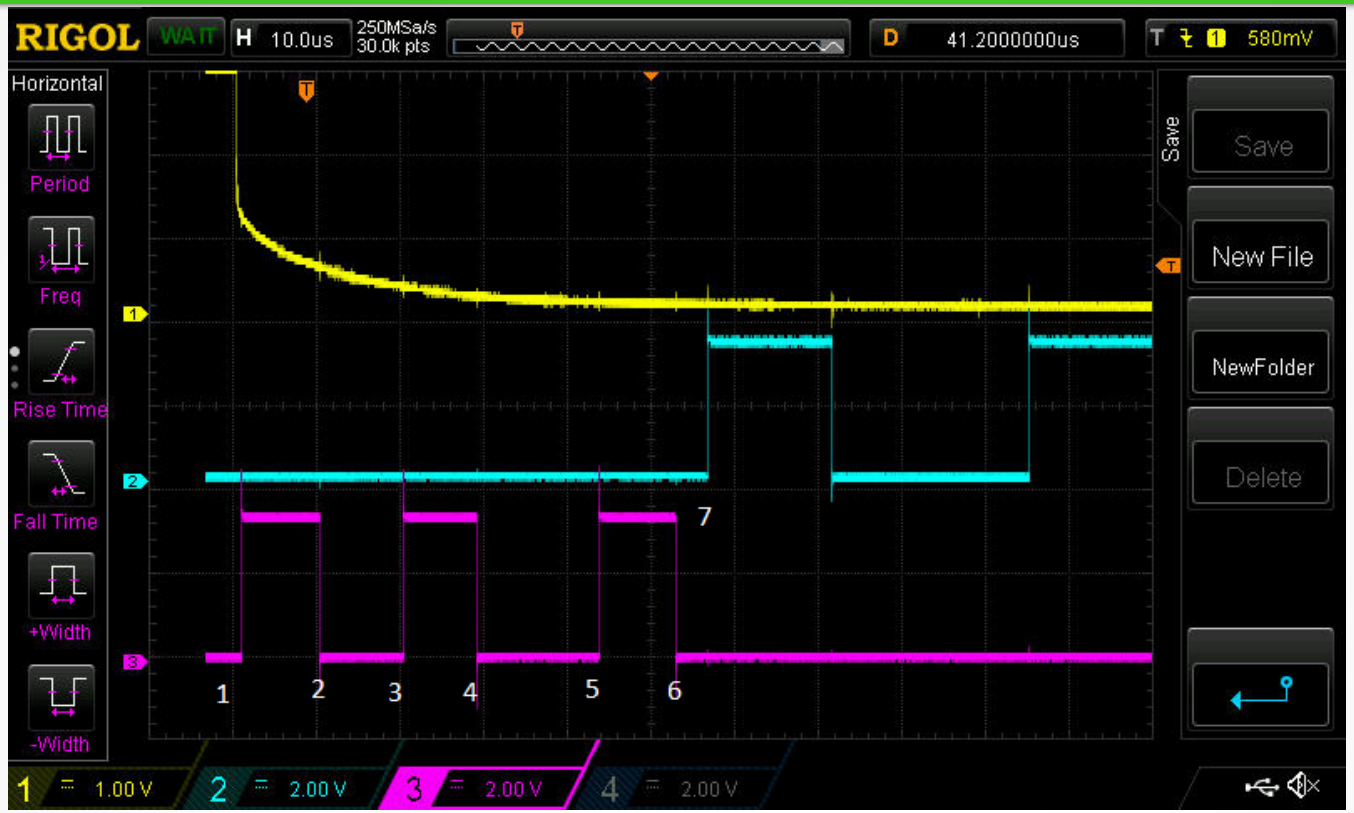
- Profiling with printf. Macros help:
  - Log entry and exit actions of every state.
  - Log events (with time-stamp) handled in every state.
  - Keep track of the current state of every HSM.
- Rely on an efficient UART output (DMA, non-blocking, zero-copy, ISR-safe). We will use it as an example when discussing optimization.
- Symbol based. Pros and cons. Compare with QSPY.
- Use GPIO for timing measuring and profiling. Example of IMU sampling. Useful for checking processor load and response time.
- CPU utilization measurement using counter in idle loop (recall UCOS).
- Keep statistics (e.g. watermark for event pools, queues, or maximum latency between interrupts). Report in console periodically or upon command.

Debugging, Profiling and Testing





Debugging, Profiling and Testing



Ch 1 User Button Ch 2 User LED (PWM) Ch 3 Debug Output

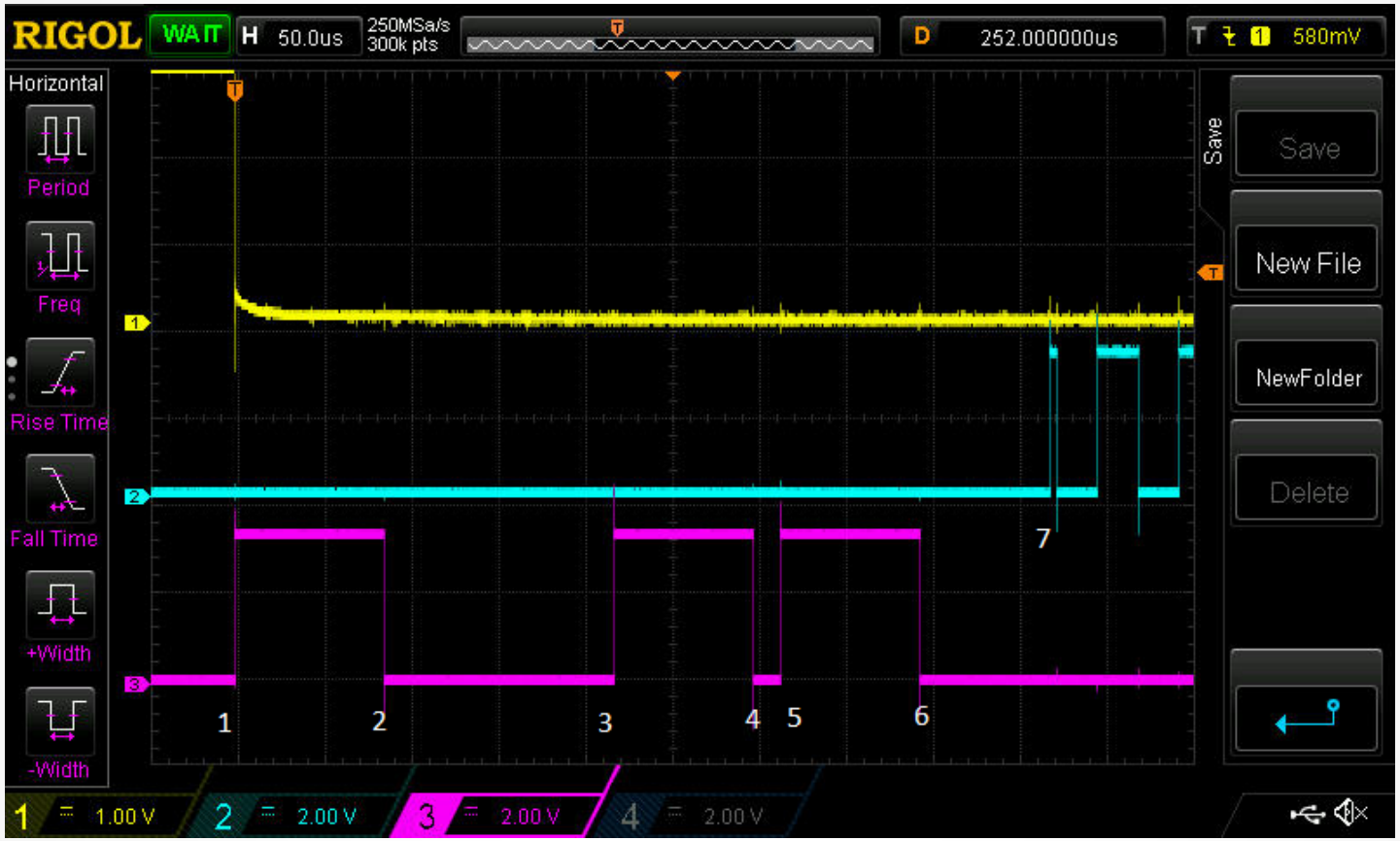
(1) Button interrupt ISR (2) USER_BTN_TRIG event in UserBtn::Up state

(3) Publishing USER_BTN_DOWN_IND (4) System publishing USER_LED_ON_REQ

(4) UserLed publishing USER_LED_ON_CFM (5) System got USER_LED_ON_CFM

(7) UserLed turns on PWM

•Debugging, Profiling and Testing



With debug print turned on, timing was changed.

Debugging, Profiling and Testing

- Testing – Most unit test framework focuses on functional model. That is the setup, testing and tear-down are done via synchronous function calls.
- They are useful for testing helper classes through their functional API. Examples include a FIFO class, a data formatting class, etc.
- But they *may* not fit the event driven model.
- What is more useful/interesting is automated integration testing. Most tricky issues arise from the interaction among asynchronous state machines, or from interacting with real hardware.

Debugging, Profiling and Testing

- Sum of testing modules individually (unit testing) !=

  Testing the sum of modules together (integration testing)
- How good is a unit test of an IMU driver that mock out the SPI bus and the real IMU chip?
- Compare to protocol tester. It needs to send a request and wait for the confirmation. It may take multiple steps.
- It also needs to verify the states of the related HSM's.
- Error injection to simulate error conditions.

Optimization

- There is not a single absolute best optimization.
- It depends on what the measurement criterion is:
  - Speed.
  - Code size.
  - Data size.
  - Power.
  - Development time.
- Ultimately it boils down to meeting requirements.
- Optimization for different criteria may contradict each other.
- Despite the title of this course, *design* and *optimization* sometimes do not agree.

Optimization vs Design

- The goal of software design is not necessarily for optimal performance. It takes into account maintainability and robustness.
- For example, a common design principle is *decoupling (separation of concerns)*. It divide the system into components (e.g. active objects).
  - Advantages: Ease of reasoning, smaller problem to solve, isolation of errors, …
  - Cost: Communication overhead (e.g. event creation and queuing), context switch (if multiple threads used), etc.

Optimization vs Design

- On the contrary, if the goal is for optimal speed performance, it may be faster to sample all sensors in a tight loop.
  - Advantages: Less overhead (e.g. no interrupt, context switching or communication overhead), better synchronization of sensor data.
  - Cost: Not scalable (adding more sensors may make the loop too long), wasteful in polling if data not available, tight coupling (e.g. different types of sensors handled together.)
- Guidelines:
  - Trade-off between an elegant architecture and performance. (Avoid going extreme to one side).
  - Optimize critical path.
  - Employ good practices (not be wasteful). Avoid over-optimizing.
  - Meet requirements.

Power – Interrupt-driven Design

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

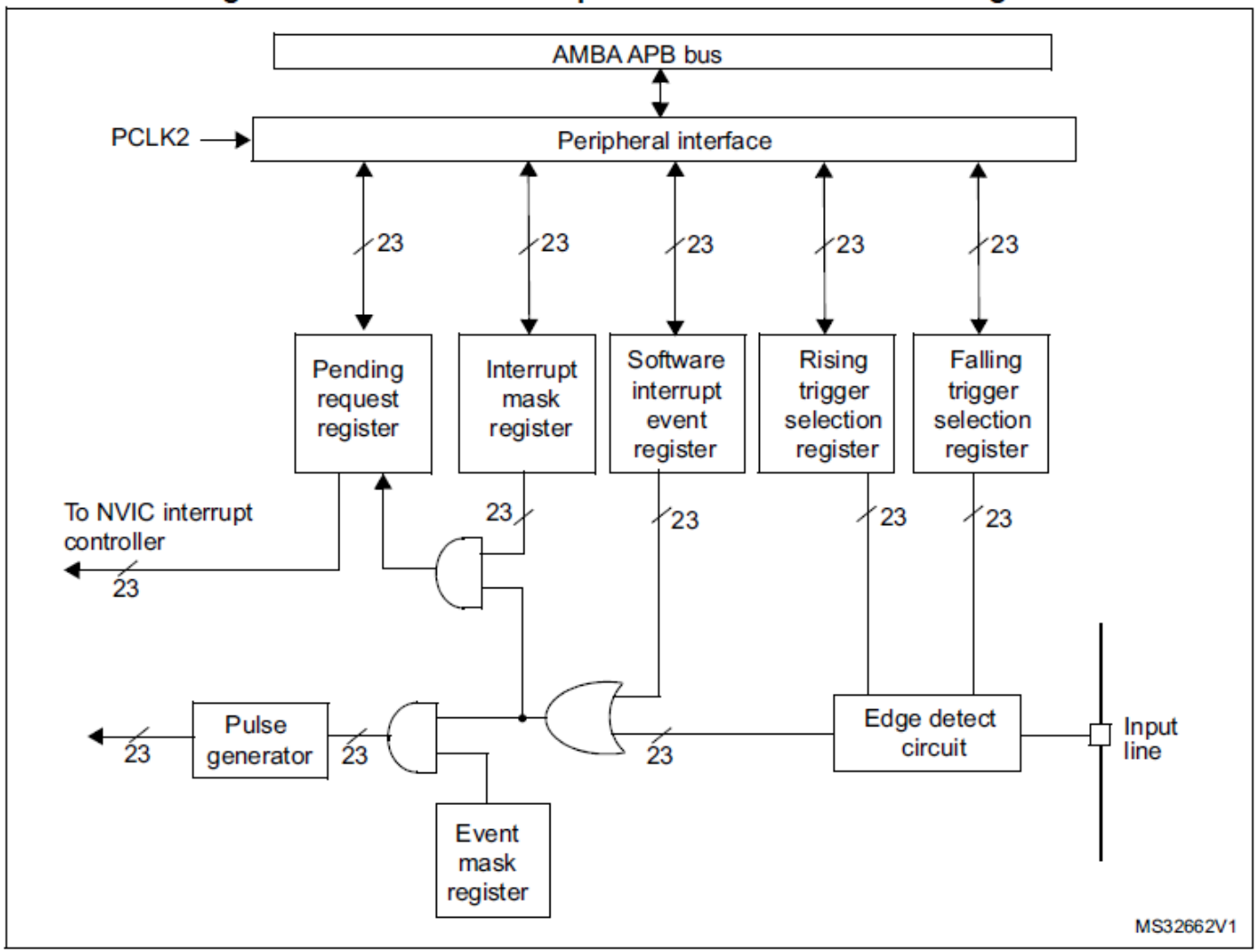
```
void QXK::onIdle(void) {
    //__WFI();   Wait-For-Interrupt
}
int_t QF::run(void) {
    QF_INT_DISABLE();
    initial_events(); // process all events posted during initialization
    onStartup(); // application-specific startup callback
    QF_INT_ENABLE();
    // the QXK idle loop...
    for (;;)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

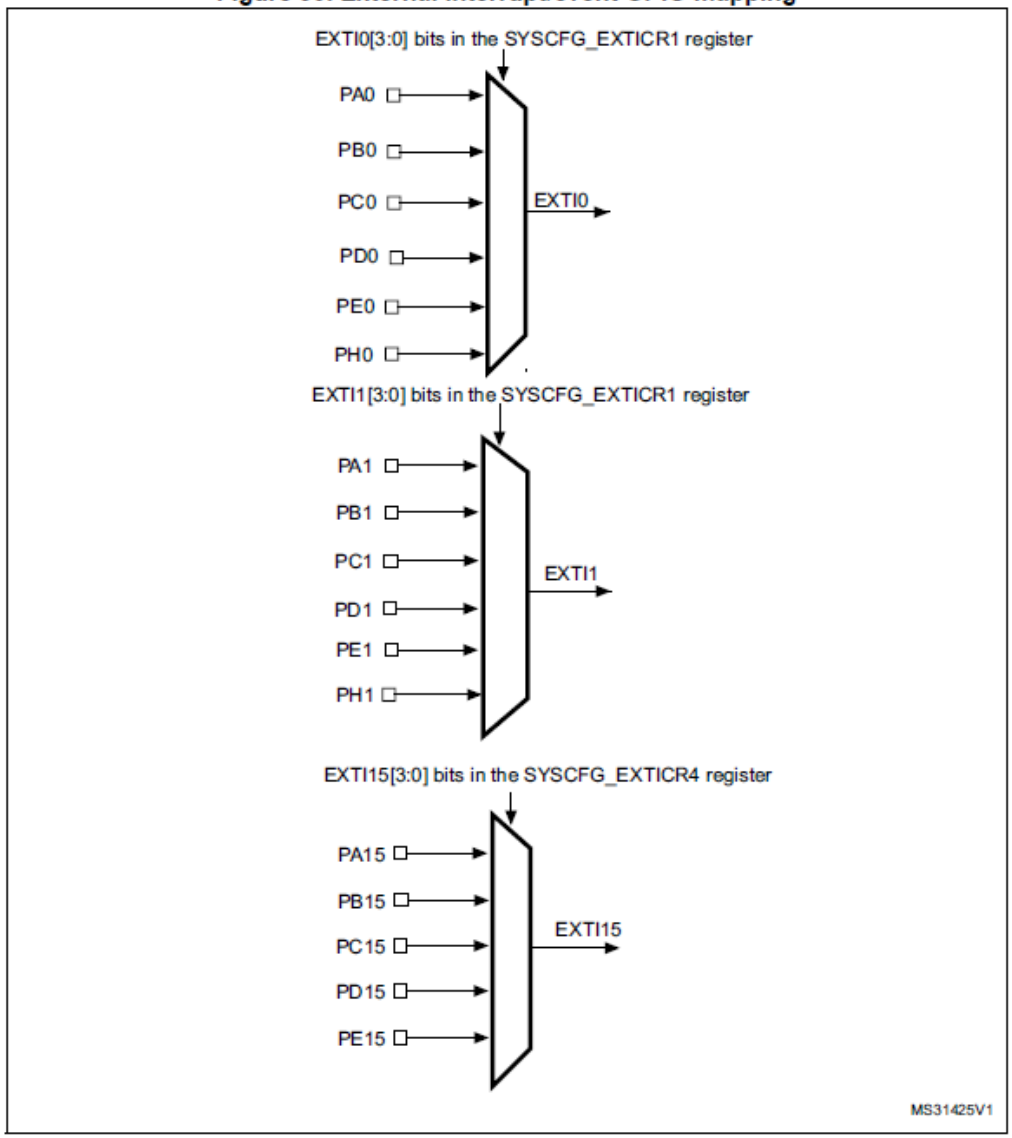
Power – Interrupt-driven Design

Figure 29. External interrupt/event controller block diagram



Power – Interrupt-driven Design

Figure 30. External interrupt/event GPIO mapping



Power – Power Management

- Need power management states.
- Best done in System.
- Shutdown hardware modules/clocks.
- Note though we want OOD, hard to do it for shared hardware blocks (e.g. DMA1 and GPIOA can be shared among different functional objects.) Manage directly in System (power states.)
- Avoid busy loop.
- Put processor to sleep mode in Idle loop of RTOS (QP or UCOS-II).

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

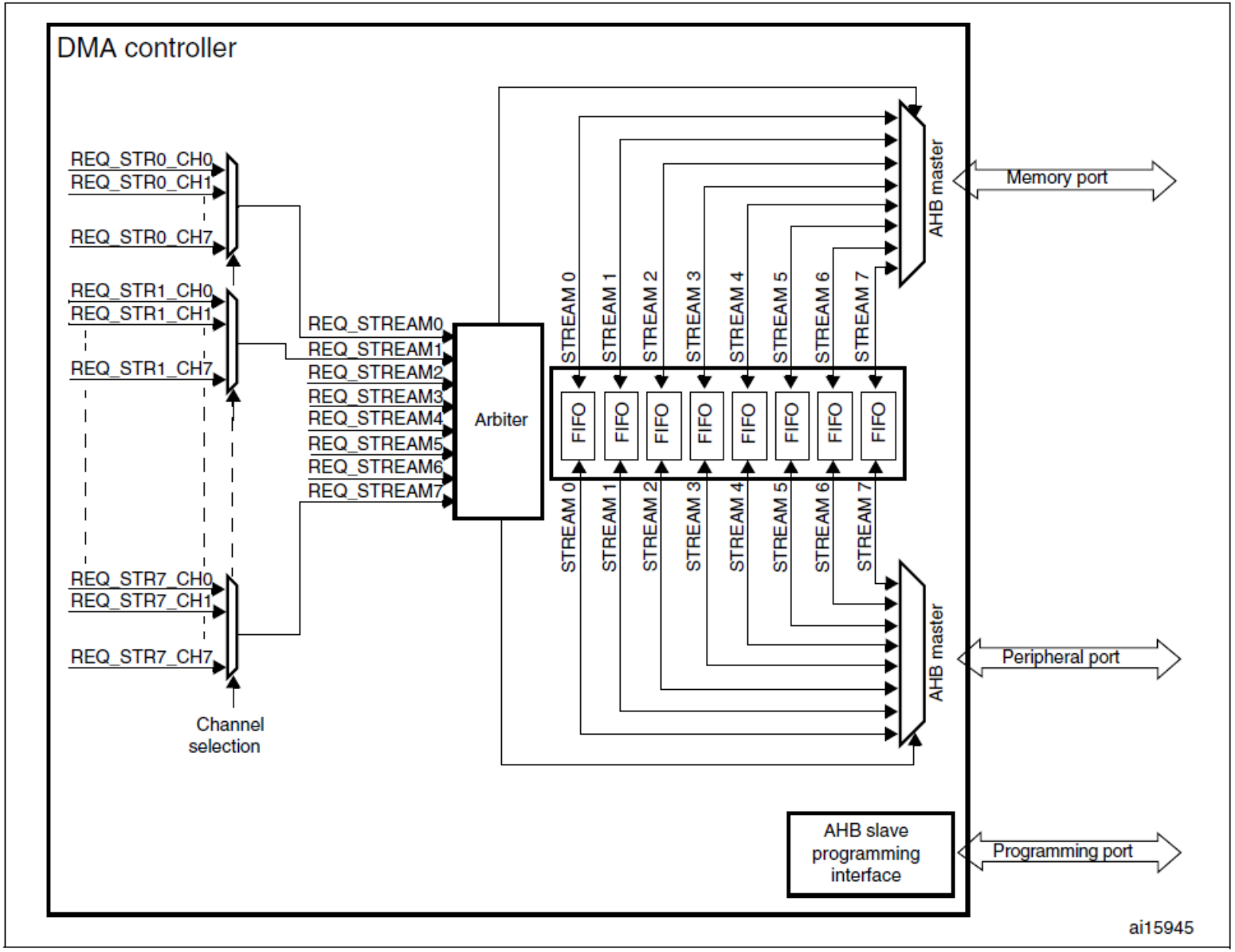
- DMA is very efficient in transferring a large amount of data between memory and peripheral (via data register).
- It can work with I2C, SPI, UART or GPIO.
- It can work in both directions (to and from peripherals).
- There is an overhead to set up each transfer, so it may not be efficient to send a 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

End of Week 7

- See you next week. Thank you.

Speed - Cache

- Cortex-M0/M3/M4 cores do not have cache. We don’t need to worry about cache with our Nucleo board.
- Cortex-M7 (e.g. STM32F769I-Nucleo board) has cache.
- See STM32F769 Programmer's Manual page 30 Section 2.2.
  - STM32F769 has 16K Data Cache and 16K Instruction Cache.
- Cache is very fast memory located between processor and main memory (e.g. internal Flash/SRAM, or external memory).
- It makes use of temporal locality principle, i.e. memory location accessed is likely to be accessed again in near future.

Speed - Cache

- Cache memory is partitioned into slots, each called a *cache line.*
- In STM32F769, cache line size is 8 words, i.e. 32-bytes. It makes use of spatial locality principle, i.e. nearby memory location is likely to be accessed at the same time. Why not make a cache line too large?
- See STM32F769 Programmer's Manual page 219 Section 4.5.3. In Table 79, see the field LineSize.
- Note line size is 2 ^(LineSize + 2). If LineSize is 1, line size is 8 words. Confirm this in Table 80.
- In Table 79 and 80, what do the terms NumSets and Associativity mean?

Speed - Cache

- Since cache size is much smaller than main memory size, we need an algorithm to map from a main memory block to cache line.
- There are three algorithms:
  - Direct mapping.
  - Associative mapping.
  - Set associative mapping.
- Direct mapping – Each main memory block is mapped to a single cache line.
  - Memory address = <Tag> <Cache line> <Word>

Speed - Cache

- *Tag* is stored along with each cache line so it knows which main memory block is currently stored in a cache line (many-to-1 mapping).
- *Cache line* (slot) selects a cache line to use.
- *Word* identifies the word (or byte) within a cache line being addressed.
- Direct mapping is simple to implement.
- However it suffers a low hit-to-miss ratio. (What is a *hit* and a *miss*?)

Speed - Cache

- Associative mapping – Each main memory block can be mapped to any cache lines.
  - Memory address = <Tag> <Word>
  - Most flexible, since any cache line can be used to hold a main memory block. Maximum hit-to-miss ratio.
  - However it is expensive to implement, as the hardware logic needs to examine tags of all cache lines in parallel.
- Set associative mapping – Compromise between direct and associative mapping. Each main memory block is mapped to a set of cache lines, i.e. it can be stored in *any* one of the cache lines in the set it is mapped to.

Speed - Cache

- Memory address = <Tag> <Set> <Word>
- *Set* is the set number the memory block is mapped to.
- For *Tag* and *Word*, see Direct mapping discussed before.
- The number of cache lines in each set is referred to as *ways*. A 2-way set associative mapping means that there are two cache lines in each set.
- Studies show 2-way set associative mapping yields significant improvement of hit ratio over direct mapping. 4-way mapping yields modest improvement.

  (Stallings, William. Computer Organization and Architecture. 4th Edition. 1996. Prentice Hall)

Speed - Cache

- What is the additional cost over direct mapping?
- Refer back to STM32F769 Programmer's Manual page 220 Section 4.5.3 Table 80.
- We now understand what it means by *NumSets* and *Associativity* along with *LineSize.*
- Let’s verify if the numbers match up:

  For STM32F769, the cache is 4-way set associative (0x3 + 1), with 128 sets (0x7F + 1). Each cache line has 8 words (2^(0x1 + 2)), i.e. 32 bytes.
- The total cache size is 4 * 128 * 32 = 16KB (correct).

Speed – Cache

- For set associative mapping, we need a cache replacement algorithm to select a cache line in a set to be replaced when it needs to make room for a new block.
  - Common algorithms:
    - Least recently used (LRU).
    - Least frequently used (LFU).
    - First-in-first-out (FIFO).
    - Random (only slightly inferior).
  - Internal to hardware implementation.

Speed – Cache

- Write policy :
  - *Write through*.
    - Memory writes update cache and main memory at the same time.
    - More bus traffic, but ensures cache and main memory always remain in sync.
  - *Write back*.
    - Memory writes only update cache. Main memory is only updated when a *dirty* (updated) cache line is replaced.
    - Less bus traffic, but cache and main memory may be out of sync.
    - A problem with DMA. Why?

Speed - Cache

- Why do we care about cache (if the processor has one)?
- Cache miss is expensive (need to read from main memory, and potentially write-back a dirty line to be replaced).
  - How could we minimize cache miss?
  - The key is to optimize for locality, such as:
    - Use static arrays over dynamic memory allocation.
    - For two dimension arrays, looping over all columns of a row is more efficient than looping over a row element of all columns.
    - For arrays of objects, group data that are frequently processed together in one class and keep others in separate classes.
    - Keep data structure size aligned to cache line size. Consider the case when a data structure spans over two cache line – may cause two replacements.
    - Avoid unnecessary multi-threading. Context switching may swap out the process stack causing many cache lines to be refilled.

Speed - Cache

- Another reason to care about cache is DMA.
- Cache coherence issue.
- Two main cases:
  - DMA read
  - DMA write
- For DMA read
  - Data is transferred by DMA from peripheral register(s) to a main memory buffer.
  - Some memory blocks of the buffer may have been cached.
  - When the processor reads from the buffer, out-of-date cached data likely will be returned.
  - How to fix? Invalidate cache lines (mapped from updated buffer) after DMA read completes. Alternatively, set the memory region containing the buffer as non-cacheable.

Speed - Cache

- For DMA write
  - Processor writes outgoing data to a memory buffer.
  - Assuming write-back policy, data are only updated in cache (unless replaced).
  - DMA is initiated to transfer data from main memory buffer to peripheral register(s).
  - Since main memory buffer may be out-of-sync with cache, outdated data are written to peripheral.
  - How to fix? Flush cache (mapped from updated buffer) before initiating DMA. Alternatively use write-through policy or set the memory region as non-cacheable.

Speed – Cache

- Cortex-M7 provides cache maintenance functions.
- See ARM Cortex-M7 Generic User Guide, Table 4-64 (page 292).
- See Table 4-67 for CMSIS functions. Note this table is incomplete. See Eclipse STM32F7 project file:
  - platform-stm32f746-disco\system\include\cmsis\core_cm7.h
- For example, to flush cache before initiating DMA write, we can call:

```
void SCB_CleanDCache_by_Addr(uint32_t *addr, int32_t dsize);
```

- See Eclipse STM32F7 project file UartOut.cpp (line 276).

Speed - Cache

- For example, to invalidate cache after completing DMA read, we can call:

```
SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize);
```

- For DMA write, an alternative method is to use write-through policy. See Eclipse STM32F7 project file main.cpp (line 215) function MPU_Config().
- Easy to visualize the effect if we miss one of the above important steps. The *part* of the UART output will simply be corrupted.
- Side note on STM32F769I Nucleo. Good upgrade if you need more processing power.

Code – Multiple Instances (HW)

- Now we move on to optimization for code space.
- More than often you would need more than one instance of the same class.
- For example in our demo project we have an active object class for the user button named UserBtn. (Similarly we have one for the user LED named UserLed.)
- It serves well to interface with the user button and user LED.
- What if we want to add more buttons and LED's?

Code – Multiple Instances (HW)

- One way is to create another active object class for the new button/LED, such as MenuBtn or StatusLed, etc.
- Let's check what will be different in each class. It turns out we will be duplicating a lot of code.
- A more optimized way is to parameterize the class such that the GPIO port/pin used for the button or LED are configuration parameters rather than hard-coded.
- The configuration parameters will be a members of the class, such as *m_port* and *m_pin*.
- Those parameters can either be passed in via the constructor (like UartAct), or contained in a static constant array of structures of configuration parameters.

Code – Multiple Instances (HW)

- What does the last point mean?
- Example:

```
typedef struct {
    uint8_t id;             // HSM ID
    GPIO_TypeDef *port;     // GPIO port
    uint32_t pin;           // GPIO pin
} LedConfig;

static const LedConfig CONFIG[] = {
    { USER_LED,    GPIOA, GPIO_PIN_5 },
    { USER_STATUS, GPIOC, GPIO_PIN_1 }
};
```

- Note that you may need more HW specific parameters (e.g. if PWM is used, you'll need to parameterize HW timer used).
- Note how a line in the structure array replaces an entire duplicated class.

Data and Speed – Non-blocking Architecture

- With QP we can encapsulate multiple HSM (regions) in a single active object.
- In essence it allows multiple HSM's to share a single *thread*.
- This is possible due to the fundamental non-blocking run-to-completion nature of HSM's.
- Question – Why is it not possible with blocking design?
- Question – What is the difference between making an HSM a region than creating its own active object?
- Less threads → Less memory overhead.
- Less threads → Less context switching overhead (faster)
- Compare to Node.js which also uses a non-blocking architecture.