EMBSYS 110, Spring 2018 Week 9

- Week 9
  - Optimization with Cache
  - Robustness and Error Handling
  - Design Heuristics
  - QP Internal Design

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

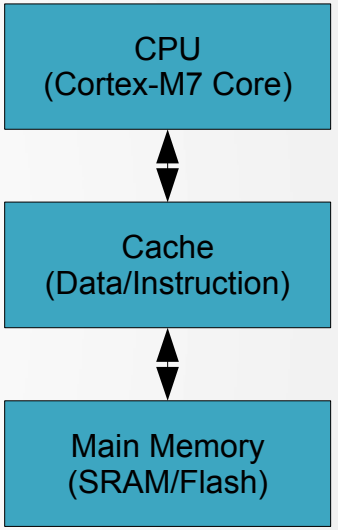
Speed - Cache

- 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 (next page).
  - 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 (next page).

Speed - Cache

Table 12. STM32F76xxx/STM32F77xxx Cortex®-M7 configuration

Features	STM32F76xxx/STM32F77xxx
Floating Point Unit	Double and single precision floating point unit
MPU	8 regions
Instruction TCM size	Flash TCM: 2 Mbytes RAM ITCM: 16 Kbytes
Data TCM size	128 Kbytes
Instruction cache size	16 Kbytes
Data cache size	16 Kbytes
Cache ECC	Not implemented
Interrupt priority levels	16 priority levels
Number of IRQ	110
WIC, CTI	Not implemented
Debug	JTAG & Serial-Wire Debug Ports 8 breakpoints and 4 watchpoints.
ITM support	Data Trace (DWT), and instrumentation trace (ITM)
ETM support	Instruction Trace interface



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. (next page)

  Note line size is 2 ^(LineSize + 2). If LineSize is 1, line size is 8 words.
- In Table 80, what do the terms NumSets and Associativity mean?

  Note values shown are one less than actual value.

  If Associativity shows 0x3, it means “4-way set associative mapping” (explained later.)

Speed - Cache

Table 80. CCSIDR encodings

CSSELR	Cache	Size	Complete register encoding	Register bit field encoding						
				WT	WB	RA	WA	NumSets	Associativity	LineSize
0x0	Data cache	4 Kbytes	0xF003E019	1	1	1	1	0x001F	0x3	0x1
		8 Kbytes	0xF007E019					0x003F		
		16 Kbytes	0xF00FE019					0x007F		
		32 Kbytes	0xF01FE019					0x00FF		
		64 Kbytes	0xF03FE019					0x01FF		
0x1	Instruction cache	4 Kbytes	0xF007E009	1	1	1	1	0x003F	0x1	0x1
		8 Kbytes	0xF00FE009					0x007F		
		16 Kbytes	0xF01FE009					0x00FF		
		32 Kbytes	0xF03FE009					0x01FF		
		64 Kbytes	0xF07FE009					0x03FF		

Speed - Cache

Tag	Cache Line (32 bytes)
	cache line 0
	cache line 1
	cache line 2
	cache line 3
	cache line 4
	cache line 5
	cache line 6
	cache line 7
	...
	...
	cache line N-1

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.

Speed - Cache

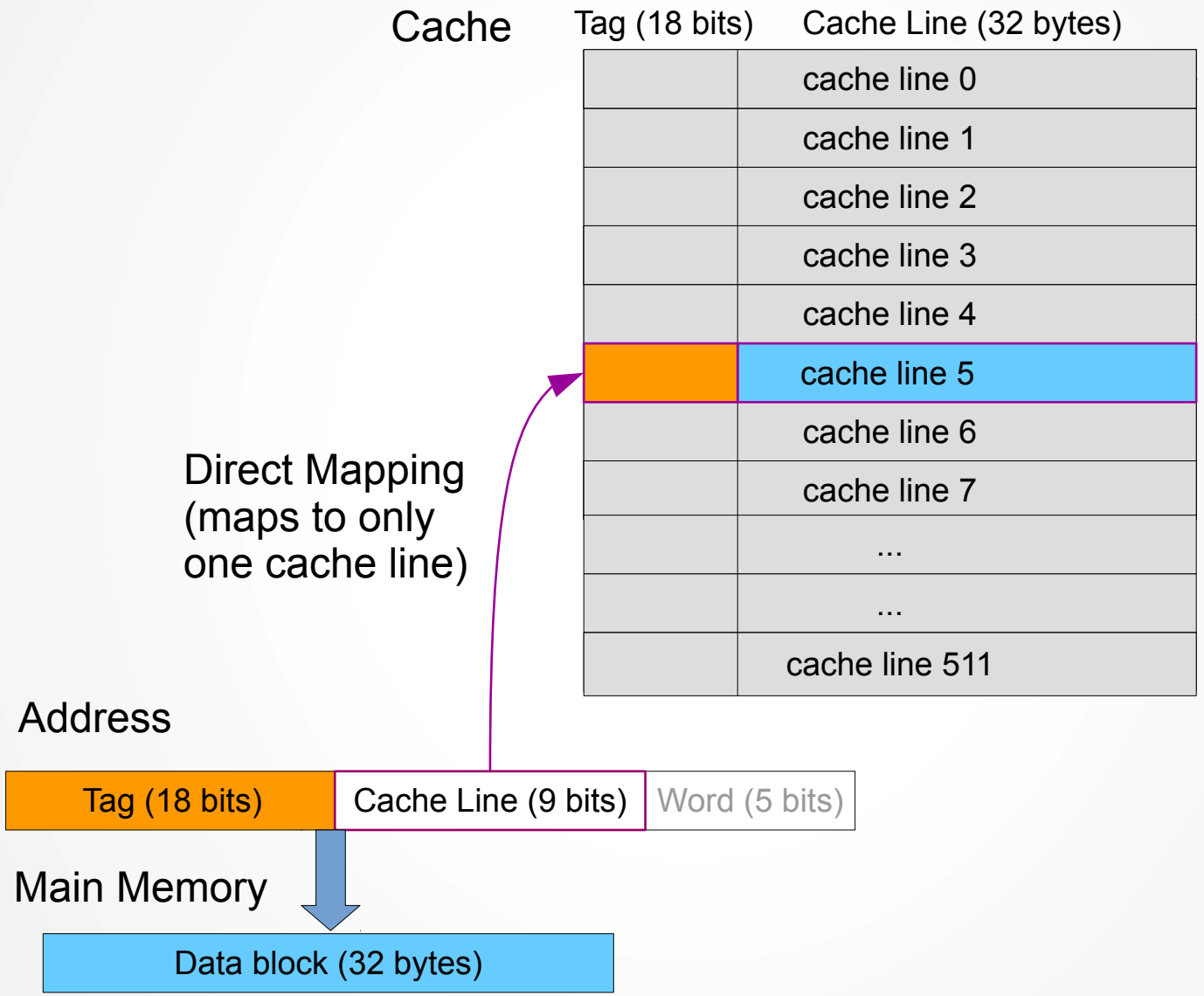
- Direct mapping – Each main memory block is mapped to a single cache line.
  - We divide a 32-bit address into 3 parts:

    Memory address = Tag | Cache Line | Word
  - Since each cache line is 32 bytes, the *Word* part needs 5 bits (2^5 = 32).
  - Since there are 512 cache lines (in STM32F769) (16KB / 32B), the *Cache Line* part needs 9 bits (2^9 = 512).
  - The *Tag* part will have the remaining 18 bits (32 – 5 – 9).

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 within a cache line being addressed.
- Direct mapping is simple to implement.
- However it suffers a low hit-to-miss ratio. (*Hit* is when a memory block is found in the cache. *Miss* is when it is not found.)

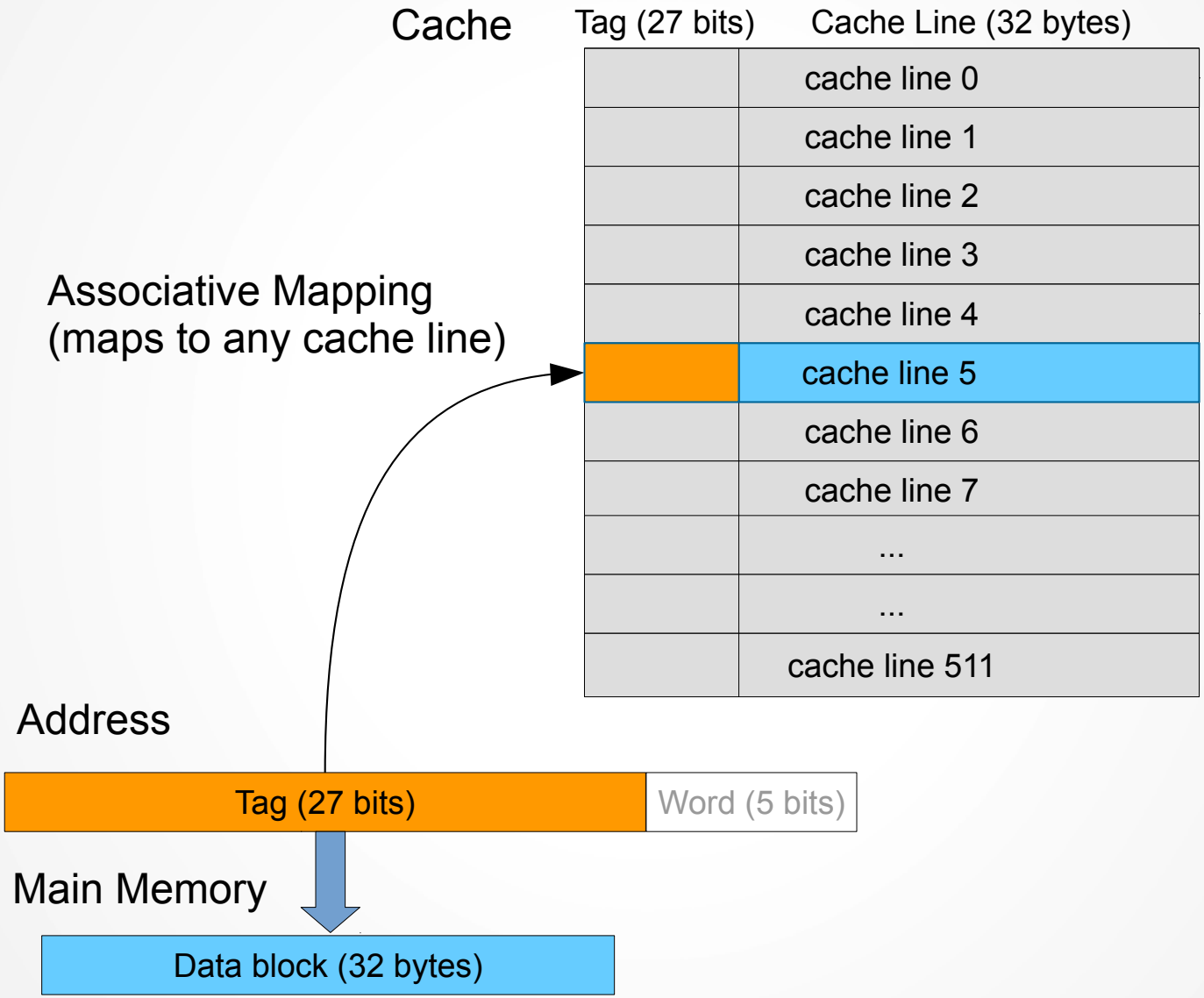
Speed - Cache



Speed - Cache

- Associative mapping – Each main memory block can be mapped to any cache lines.
  - Memory address = Tag | Word
  - *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).
  - *Word* identifies the word within a cache line being addressed.
  - 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.

Speed - Cache



Speed - Cache

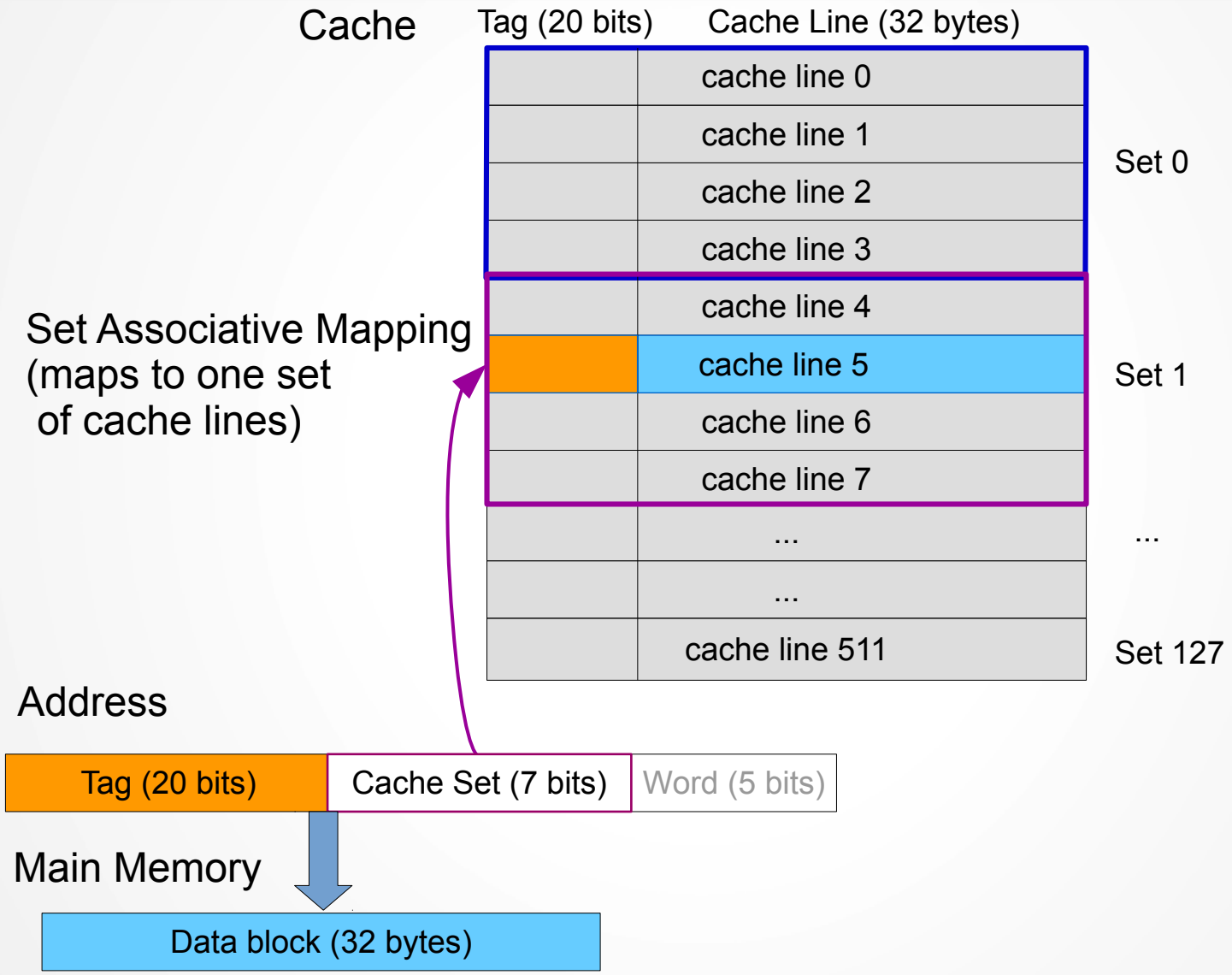
- Set associative mapping – Compromise between direct and associative mapping. Each main memory block is mapped to a set of cache lines, i.e. it can be stored in *any* one of the cache lines in the set it is mapped to.
- Memory address = Tag | Cache Set | Word
  - *Cache Set* is the set number the memory block is mapped to.
  - *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).
  - *Word* identifies the word within a cache line being addressed.

Speed - Cache

- 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



Speed - Cache

- What is the additional cost of set associative 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).

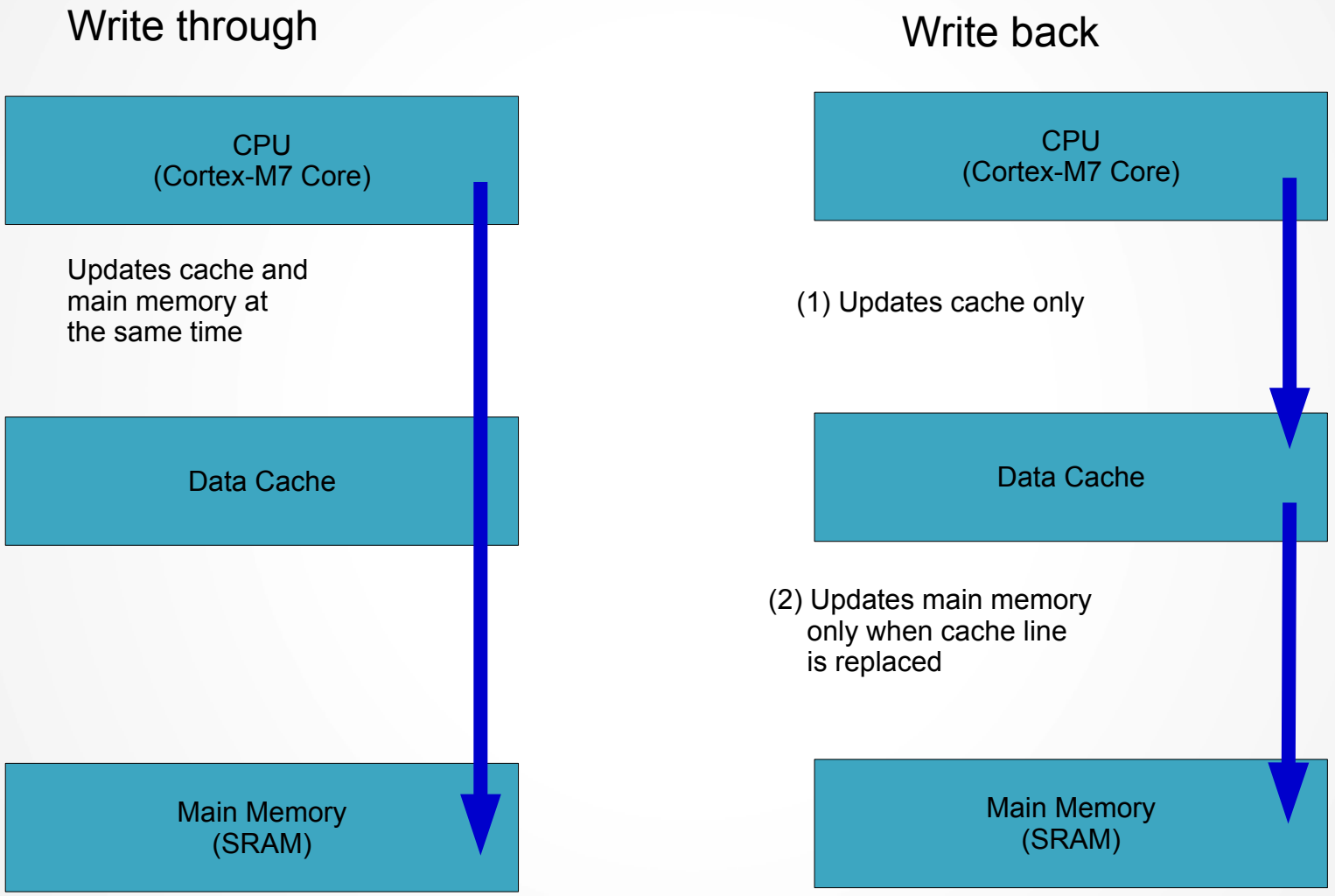
Speed – Cache

- 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



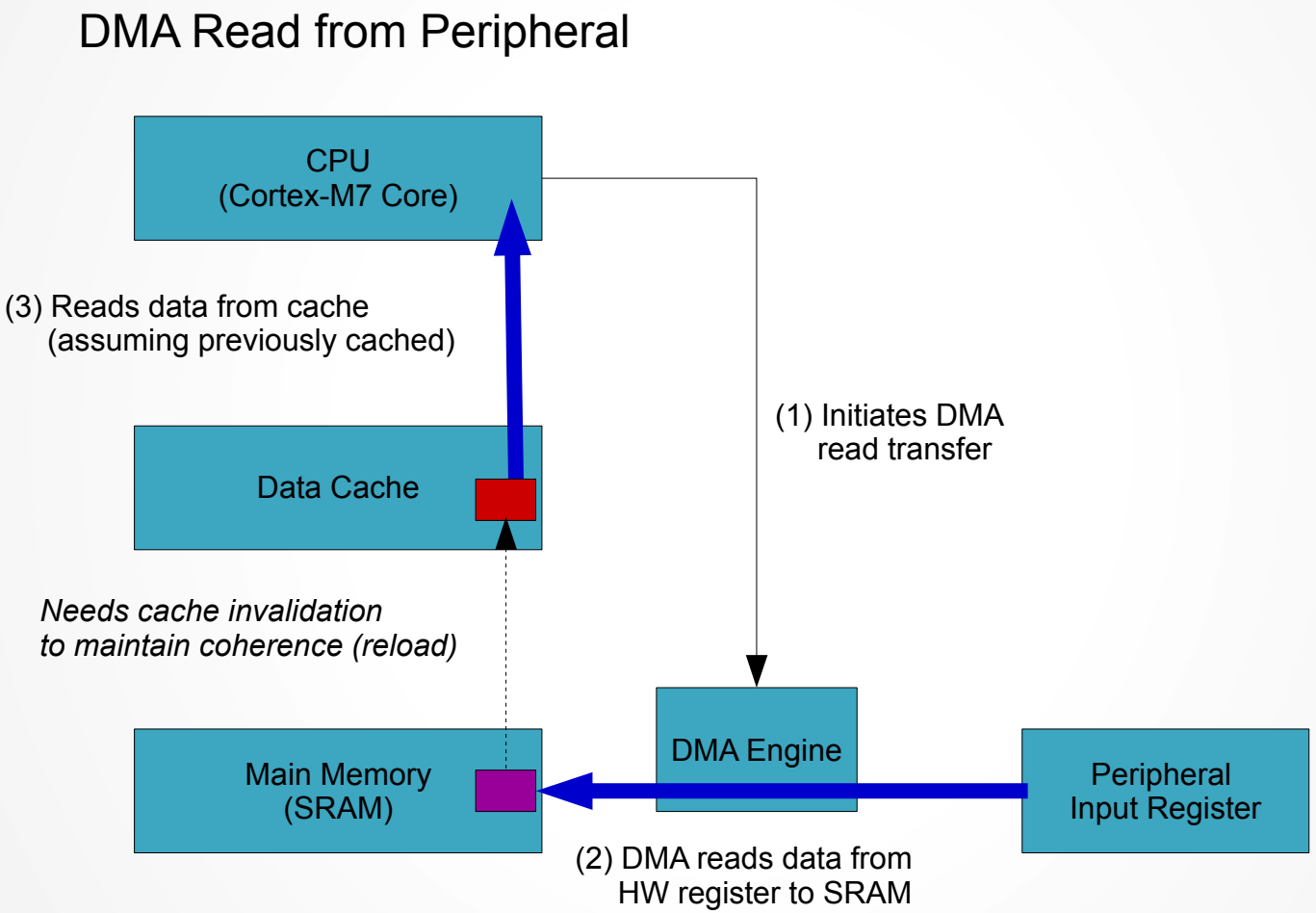
Speed - Cache

- 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 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

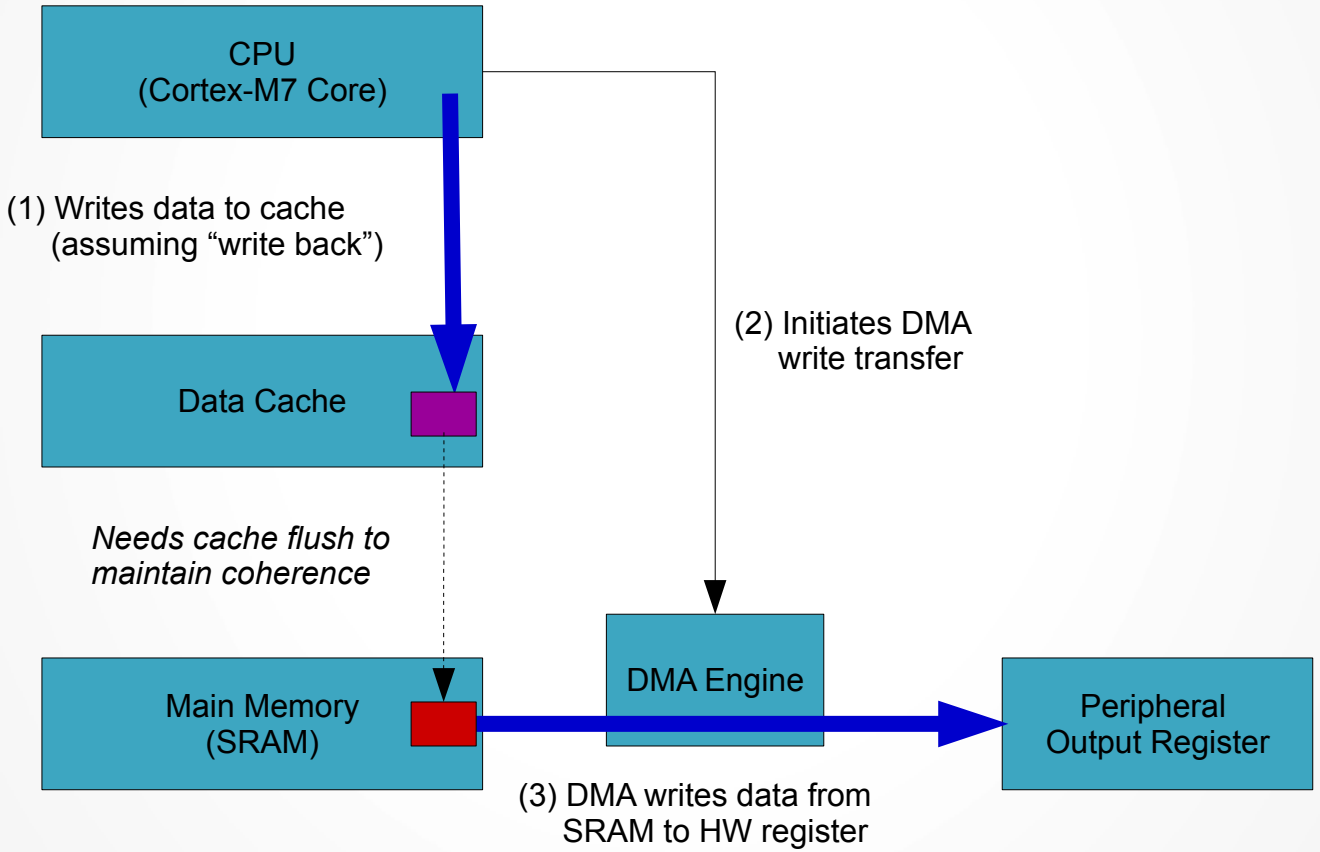


Speed - Cache

- 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

DMA Write to Peripheral



Speed – Cache

- Cortex-M7 provides cache maintenance functions.
- See ARM Cortex-M7 Generic User Guide, Table 4-64 (page 292).
- See Eclipse STM32F7 project file “system\include\cmsis\core_cm7.h”.
- Examples of CMSIS data cache functions:

  void SCB_EnableDCache (void);

  void SCB_DisableDCache (void);

  void SCB_InvalidateDCache (void);

  void SCB_CleanDCache (void);

  void SCB_CleanInvalidateDCache (void);

  void SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize);

  void SCB_CleanDCache_by_Addr (uint32_t *addr, int32_t dsize);

  void SCB_CleanInvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize);

Speed - Cache

- To invalidate cache after DMA read, call:

  void SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize);

- To flush cache before DMA write, call:

  void SCB_CleanDCache_by_Addr(uint32_t *addr, int32_t dsize);

  Example:

```
Fifo &fifo = *(me->m_fifo);
uint32_t addr = fifo.GetReadAddr();
uint32_t len = fifo.GetUsedCount();
if ((addr + len - 1) > fifo.GetMaxAddr()) {
    len = fifo.GetMaxAddr() - addr + 1;
}
FW_ASSERT((len > 0) && (len <= fifo.GetUsedCount()));
//SCB_CleanDCache();
SCB_CleanDCache_by_Addr((uint32_t *)(ROUND_DOWN_32(addr)),
                        ROUND_UP_32(addr + len - ROUND_DOWN_32(addr)));
HAL_UART_Transmit_DMA(&me->m_hal, (uint8_t*)addr, len);
```

Speed – Cache

- For DMA write, an alternative method is to use write-through policy. See Eclipse STM32F7 project file main.cpp (line 215) function MPU_Config().

```
static void MPU_Config(void) {
  MPU_Region_InitTypeDef MPU_InitStruct;
  HAL_MPU_Disable(); // Disable MPU
  // BaseAddress must be aligned to multiples of size. For the STM32F746 Discovery board
  // it has 320KB SRAM. For cover it all, we have to select 512KB and start from 0x20000000.
  // Note the original example use the base address of 0x20010000 and the size of 256KB. This is
  // incorrect since 0x20010000 is only aligned to 64KB and not 256KB. As a result, the range
  // covered is actually from 0x20000000 to 0x20040000 (256KB only, missing the last part).
  MPU_InitStruct.Enable = MPU_REGION_ENABLE;
  MPU_InitStruct.BaseAddress = 0x20000000;              // was 0x20010000.
  MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;          // was MPU_REGION_SIZE_256KB.
  MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
  MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;// “not sharable” forces write-through.
  MPU_InitStruct.Number = MPU_REGION_NUMBER0;
  ...
  HAL_MPU_ConfigRegion(&MPU_InitStruct);
  HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT); // Enable MPU
}
```

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.

Robustness

- A software system is robust when it can cope with not just normal situations but also unexpected ones.
- One that crashes upon unexpected user input or network packets is NOT robust. (For example, my laptop crashes when I close the lid and reopen it quickly! Other examples?)
- One that hangs or simply becomes unresponsive for a long time is NOT robust. (Example?)
- How about one that handles unexpected situations in an ad-hoc (or not precisely defined) manner? A “low level programmer” (recall Harel’s paper) may end up making decisions on whether *or* how to handle an error status returned from a called function, which can be retrying for N times, using different parameters, or simply returning an error.

Robustness

- Since there may be little coordination among programmers about these “low-level” details (and even if there is it may not be clearly defined), the overall or global system reaction to such unexpected situations is usually not easy to figure out. (What happens when I reopen the lid quickly?)
- Discussion – How about C++ exception? (try, catch, thow, etc.) See PSCiCC.
- An embedded real-time system (a.k.a. reactive system) has to be robust, since by definition it has to react to all kinds of asynchronous input from the “world”, which can be in any order, with various timings, expected or not.

Error Handling

- Statechart is a nature fit for specifying error handling. Why?
- Statechart **can** *precisely*, *concisely* and *completely* describe system behaviors, which can include how it reacts to error situations.
- Note the word "**can**" above. A designer (you) still have to think about any possible error situations and how they should be handled.
- What's different now?

Error Handling

- Differences:
  - Instead of thinking about error handling when you are coding, you think about it when doing a proper design.
  - You can focus on logical behaviors (or interactions) rather than coding syntax.
  - A statechart shows error handling behaviors explicitly and allows them to be easily visualized and communicated.
  - Hierarchical states make default error handling easy.
  - State transition makes it easy to “jump” back to retry or restart. (Compare to loops.) In other words it is flexible to control the flow.

Error Handling

- Entry and exit actions guarantee proper initialization and cleanup (synchronous). Important to ensure resources are freed when exiting a state due to errors. Once specified in an an exit action, it applies to all cases when a state is exit. (Think about calling “fclose()” on every error.)
- Timer is convenient to use. (Compare to checking tick count in a loop.) Typical pattern is:
  - Starting timer when entering wait state.
  - Stopping timer when exiting from wait state.
  - Handling timeout event in wait state.

Error Handling

- PWS and event semantics ensure safety constraints are met. Examples:
  - When exiting an operational state (e.g. Scanning) in the ***whole*** layer, it is guaranteed all ***parts*** (e.g. Motor, RadioBeam, etc) are shutdown.
  - How? Via STOP_REQ and matching CFM.
  - After stopping all parts, it is guaranteed all resources previously allocated for them will not be accessed any more and will be safe to delete (a shared FIFO, file handle, etc.)
- Asynchronous non-blocking event-driven paradigm allows any exception conditions to be detected and handled as soon as *required*.

Error Handling Strategies - Assert

- Decide which errors to handle and which *not*.
  - *"Not"* does not mean to ignore the error.
  - It means firing an *assert*, which typically causes reboot.
  - It should also log an error.
  - Reboot is a drastic but effective measure to recover from an otherwise unrecoverable error.
  - Reboot is better than having the system run erratically or being unresponsive. It brings the system to fail-safe state.

Error Handling Strategies - Assert

- Assert is to verify a condition or assumption that must be true unless there is a software bug or an *unlikely* hardware malfunction.
- This is called “design by contract”. See PSCiCC.
- Discussion – Does a hardware malfunction trigger assertion? Yes, it simplifies design (less error cases to handle). No, it should continue to run to for diagnosis. Consider external components vs internal blocks.
- Discussion – Do you disable assert for production release?

Error Propagation

- Do not assert for these cases:
  - Invalid user input.
  - Invalid data packet.
  - Error status from another software components (CFM/IND event with failure). Treat them as black boxes. Though they may not fail today, they may fail in the future. Make no assumption.
  - Invalid REQ events from another software components. Same rational as above. It includes REQ arriving in an unexpected or inconvenient state. (asynchronous system).
  - Error conditions detected in certain hardware components (e.g. external ones that are more likely to fail).

Error Propagation

- How to handle errors that do not trigger assert?
  - Retry. Upon failed CFM from the lower layer (the *part*), retry for a number of times.
  - Propagation. Return failed CFM to the upper layer (the *whole*) with reason and source of failure.
  - Report the error to the user which can be a human operator or another machine on the network. Examples include resource not available, hardware failure, etc. It is up to the external user to decide how to handle it (retry, diagnosis, etc.)
  - Handle the error as explicitly defined in the statechart (app specific).
- The above is done iteratively until the error event is propagated to the topmost state machine.

Case Study - UartAct

- *UartAct* stands for UART Active Object.
- It is a *part* to System. It is a *whole* of the composed UartIn and UartOut (parts).
- Note the use of timers in wait states.
- Note the propagation of start and stop errors to the upper layer.
- Check start and stop requests are handled in all states.
- When UartAct returns to the Stopped state, it is guaranteed UartIn and UartOut are both stopped as well. Safe to free shared resources such as FIFO’s.

Case Study - UartAct

- Design review
  - What is the purpose of UART_OUT_FAIL_IND and UART_IN_FAIL_IND?
  - Are they handled in all possible states?
  - If not (e.g. starting), how to fix it?
  - (defer...)
- A subtle but important note on the asymmetry between starting and stopping paths:
  - It favors stopping for fail-safe design. A stop request is expected to always succeed (assert otherwise).
  - A stop request is automatically deferred if it cannot be conveniently be handled in a transient state.

Case Study - UartAct

- A start request may fail (as discussed above). It can be handled with retries, propagation, or by reporting to the end user.
- Example of start request failure – rapid stop-start requests back-to-back. The state machine is in Stopping state when it receives a start request.
- How would you handle it? It can be in the middle of stopping others components.
- A simple way is to defer it like stop request. However we don’t want to defer both stop and start request. Why?
- Here we just return failure (let upper layer retry). In reality it shouldn’t happen if the upper layer (whole) always waits for stop CFM before restarting (no rapid stop-start sequence above). Again we are extra cautious as we are not making assumption!
- Example – Console command to restart entire system.

Design Heuristics

- We adopted the following common semantics:
  - REQ and CFM (request and confirmation)
  - IND and RSP (indication and response)
- They form pairs with matching sequence numbers.
  - CFM acknowledges REQ.
  - RSP acknowledges RSP.
- Question – why use sequence number? (Race conditions.)
- Acknowledgment is like returning status from a function. The norm is to check with exception allowed. (Not the other way round!)

Design Heuristics

- Each HSM exposes an event interface. It describes the services it provides.
- How an service is fulfilled is hidden. Only care about the results. (Just like… :)
- Max timeout is part of the REQ. Otherwise how does the requesting object know how long to wait? Guaranteed to send a confirm within max timeout.
- Upon timeout, requesting object should handle the timeout exception case (as an internal event).
- Redundancy in timeout handling? Done in both requesting and requested object. Why is it necessary? Different teams work on different components. No assumption, no trust. Why does a computer program hang?

Design Heuristics

- When designing a statechart, it is important to handle every request in every state! (So for some indications as well.)
- It sounds challenging. But thanks to state hierarchy!
- What if you missed one state? Thanks to timeout in requesting object! Now we see the importance of redundancy.
- Examples of the need to handle a request in an unexpected state? Actually a common cause for race conditions! Statechart exposes all these cases so we can see them! (People say statechart makes a design complex, but the fact is...)

Design Heuristics

- Note the use of internal events as reminders. Useful for
  - Breaking up a long chain of actions.
  - Factorizing common actions.
  - Just to make a statechart look neater!
- Protocol reference. Q.921 LAPD/HDLC and BLE.
- Summary of statechart design pattern so far
  - Use well-defined event interface (REQ/CFM/IND/RSP).
  - Use sequence number.
  - Always check for timeout.
  - Handle all requests in all states.

QP Internals

- References

  [1] Samek, Miro. Practical Statecharts in C/C++, 2nd Edition. Newnes. 2009.

  [2] Quantum Leaps, LLC. Application Note QP and ARM Cortex-M. 2016.

- See Figure 2. block diagram in App Note [2] (page 2).
- It contains

  (1) A kernel (QV, QK or QXK).

  (2) An event processor (QEP) based on hierarchical state machine (Qhsm).

  (3) An event-driven framework (event pools, event queues, timers, publish-subscribe mechanism).

- While (3) is straighforward, (1) and (2) are tricky.

QXK

- QXK is the *extension* of QK (Quantum Kernel).
- QXK is a preemptive RTOS, just like UCOS-II.
- There are two types of threads:
  - Basic threads (a.k.a. AO thread).
  - Extended threads (can block).
- The original QK only supports basic threads which *cannot* block. Each thread is an active object that processes events in run-to-completion steps.
- Being non-blocking means that it is truly asynchronous.

QXK

- Note on asynchronous design:
  - Asynchronous vs synchronous.
  - Callback vs events.
- Being non-blocking means the highest priority task (i.e. thread/active object) always run to completion.
- Simplifies context-switching design, allowing the use of a single stack (main stack) vs one for each task (process stack).
- When does context-switching occur in RTOS?

QXK

- Answer:
  - When an RTOS API is called, which makes a higher priority task *ready*.
  - When an interrupt occurs, which makes a higher priority task *ready*.
- In traditional RTOS, the ways to make a higher priority task ready include unlocking a mutex, signaling a semaphore, posting to a mailbox, setting an event flag, timer expiring, etc.
- Do you really need all these different kinds of IPC (inter-process communication)? Think about where a task may block...
- QP argues that you only need one – event queues. Single point of blocking per task. Unified IPC.
- That's what QK supports. QXK extends it by adding semaphore, etc.

QXK

- See Figure 5. block diagram in App Note [2] (page 18).
- It illustrates the 2nd case of context-switching caused by interrupts.
- For the 1st case of context-switching caused by API calls (posting events) there are no interrupts involved (PendSV or NMI), using just function calls. More efficient. See qpcpp\include\qxk.h.

QXK

```
#define QACTIVE_EQUEUE_SIGNAL_(me_) do { \
    QXK_attr_.readySet.insert((me_)->m_prio); \
    if (!QXK_ISR_CONTEXT_()) { \
        if (QXK_sched_() != static_cast<uint_fast8_t>(0)) { \
            QXK_activate_(); \
        } \
    } \
} while (false)
```

- Take a look at QXK_sched_() and QXK_activate_() in qpcpp\source\qxk.cpp.
- Focus on basic thread to basic thread context-switch.

QXK

- For the 2nd case of context-switch caused by interrupts, see macro in qpcpp\ports\arm-cm\qxk\iar\qxk_port.h:

```
#define QXK_ISR_EXIT()  do { \
    QF_INT_DISABLE(); \
    if (QXK_sched_() != static_cast<uint_fast8_t>(0)) { \
        QXK_CONTEXT_SWITCH_(); \
    } \
    QF_INT_ENABLE(); \
} while (false)
```

- See QXK_CONTEXT_SWITCH_() in qxk_port.h. Note that all it does is to trigger the PendSV interrupt.

QXK

- Let's look at *PendSV_Handler* in qpcpp\ports\arm-cm\qxk\iar\qxk_port.s.
- Note the different types of context-switch:
  (1) Basic to basic thread (single stack, using MSP).
  (2) Extended to extended threads (individual stack for each thread, using PSP).
  (3) Basic to extended thread.
  (4) Extended to basic thread.
- Type (2) above is just like traditional RTOS, like UCOS-II.
- Type (1) above is good for RTC active objects.

QXK

- For extended threads (type (2)), see the following functions:
  - QXK_stackInit_fill – Initialize stack to switch to a thread handler function for the first time (assumes no FP state).
  - PendSV_save_ex – Save the complete context to the current process stack (supports FP lazy stacking).
  - PendSV_restore_ex – Restore the complete context from the next process stack.
- Note hardware automatically save part of the stack frame to the current process stack. See Figure 2-3 (page 2-27) of ARM Cortex-M4 Generic User Guide.
  - Why does it only save R0-3, R12, LR, PC and xPSR?
  - Why does it need to save the rest in PendSV_save_ex?

QXK

- For basic threads (type (1)), see the following functions:
  - PendSV_activate – Fall thru from PendSV_Handler. It is very tricky that it constructs a new stack frame so that it can return to the C function QXK_activate_() to activate the next (highest priority) active object.

    Why does it NOT save the rest of the context (i.e. R4-R11, etc)? Think about APCS. They are automatically saved by the C function when they are used!
  - Thread_ret – Return to here from QXK_activate_() where there are no more higher priority active objects. Tricky how this return address is set up in the artificial stack frame above.
  - NMI_Handler – Artificial interrupt to go back to handler mode, so it can return from the original interrupt back to the preempted thread. It is tricky how it discard the NMI stack frame (in main stack) to expose the original stack frame (pushed by the original interrupt) (again in main stack).