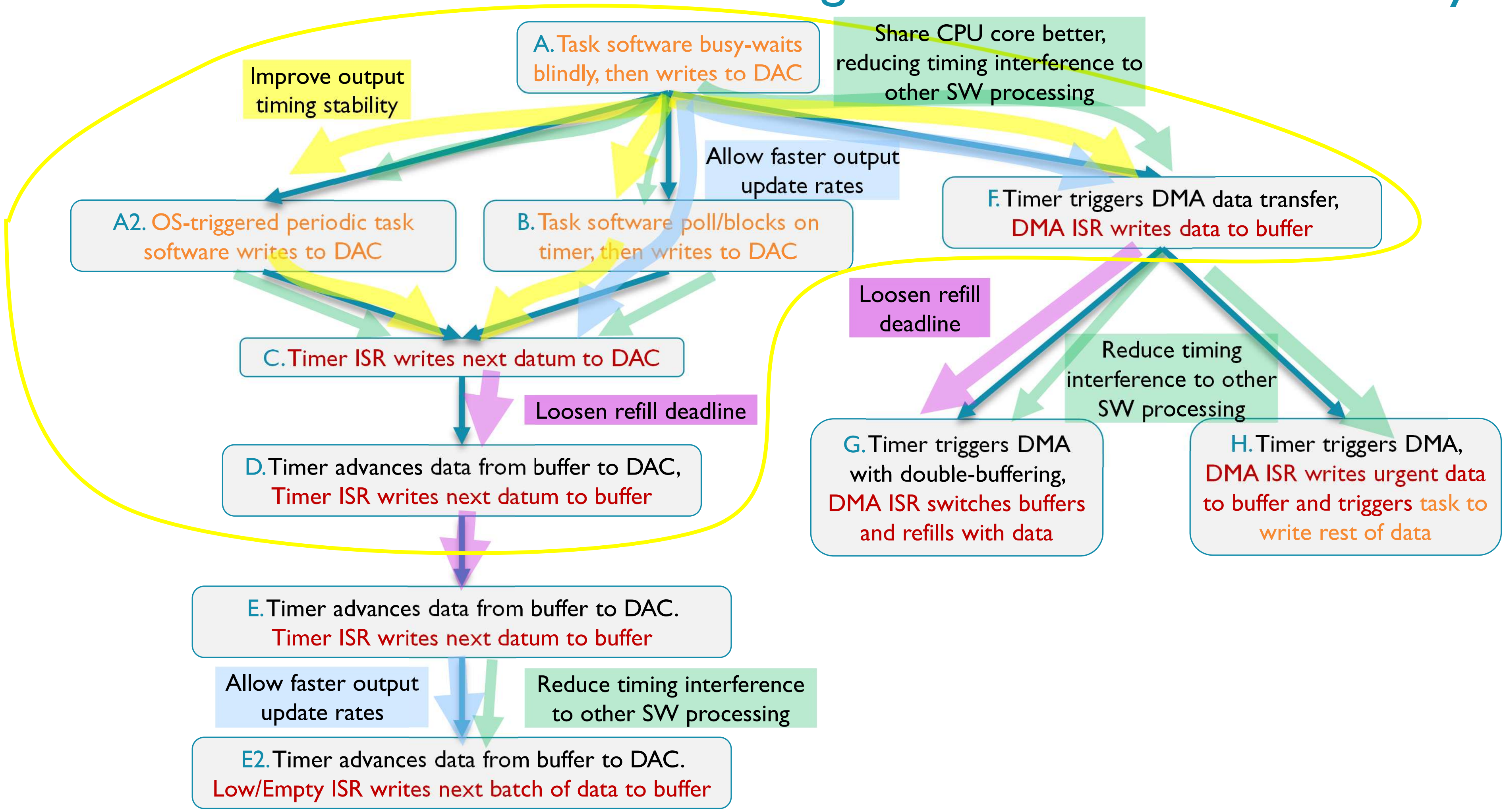
NC STATE UNIVERSITY

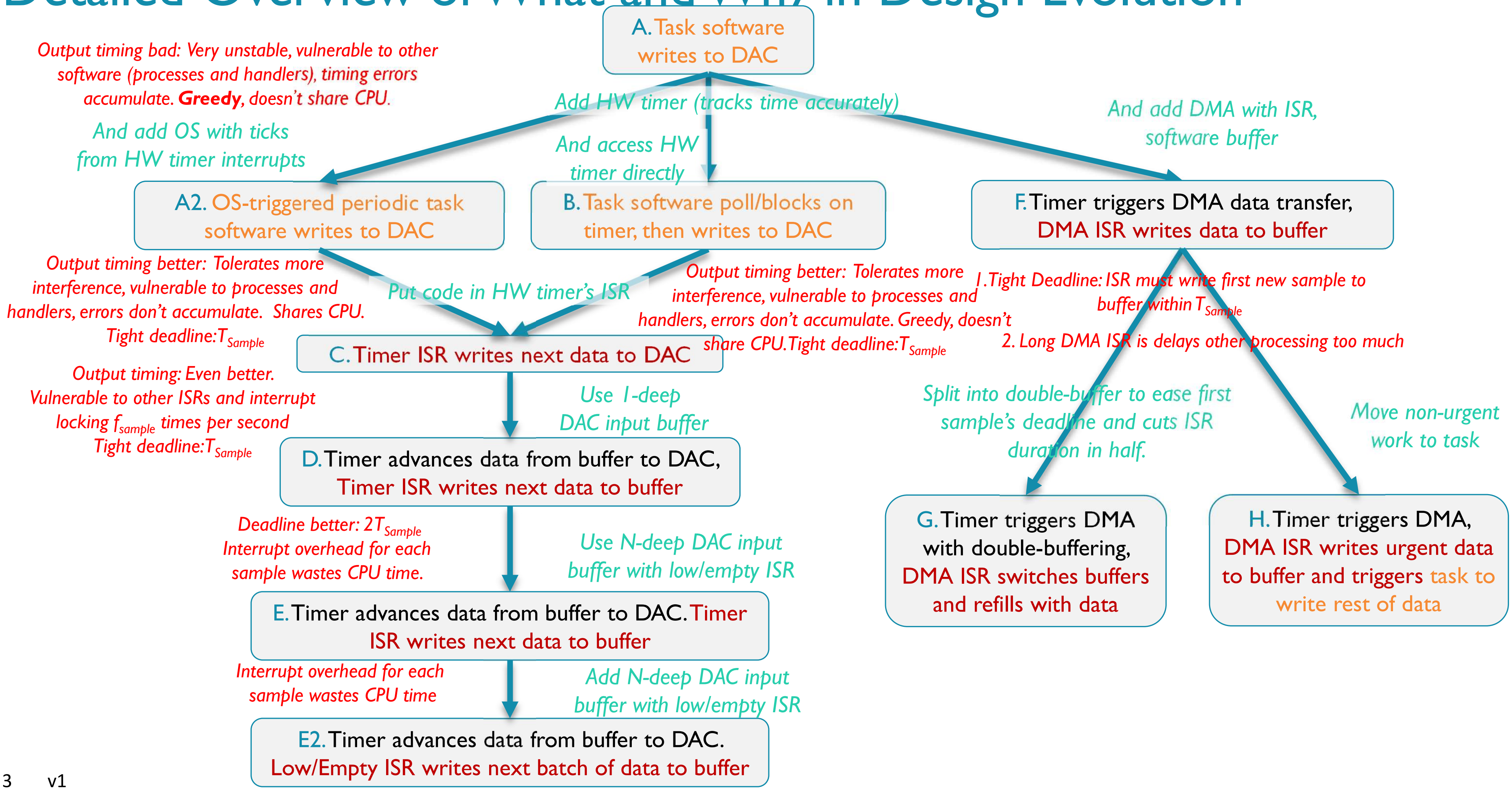
16: Improving Output Timing Stability for WaveGen (Part 2)

v1

Overview of Waveform Generator Design Evolution: What and Why

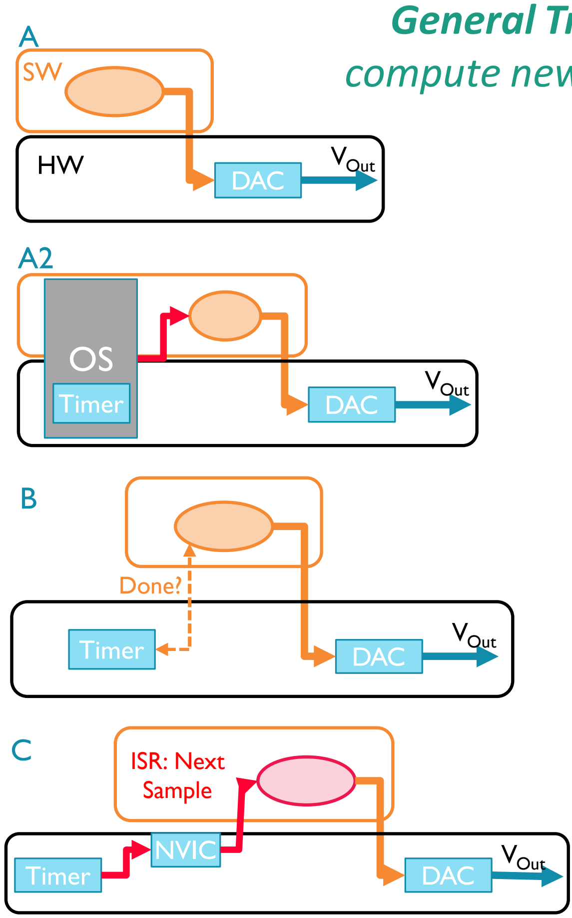


Detailed Overview of What and Why in Design Evolution

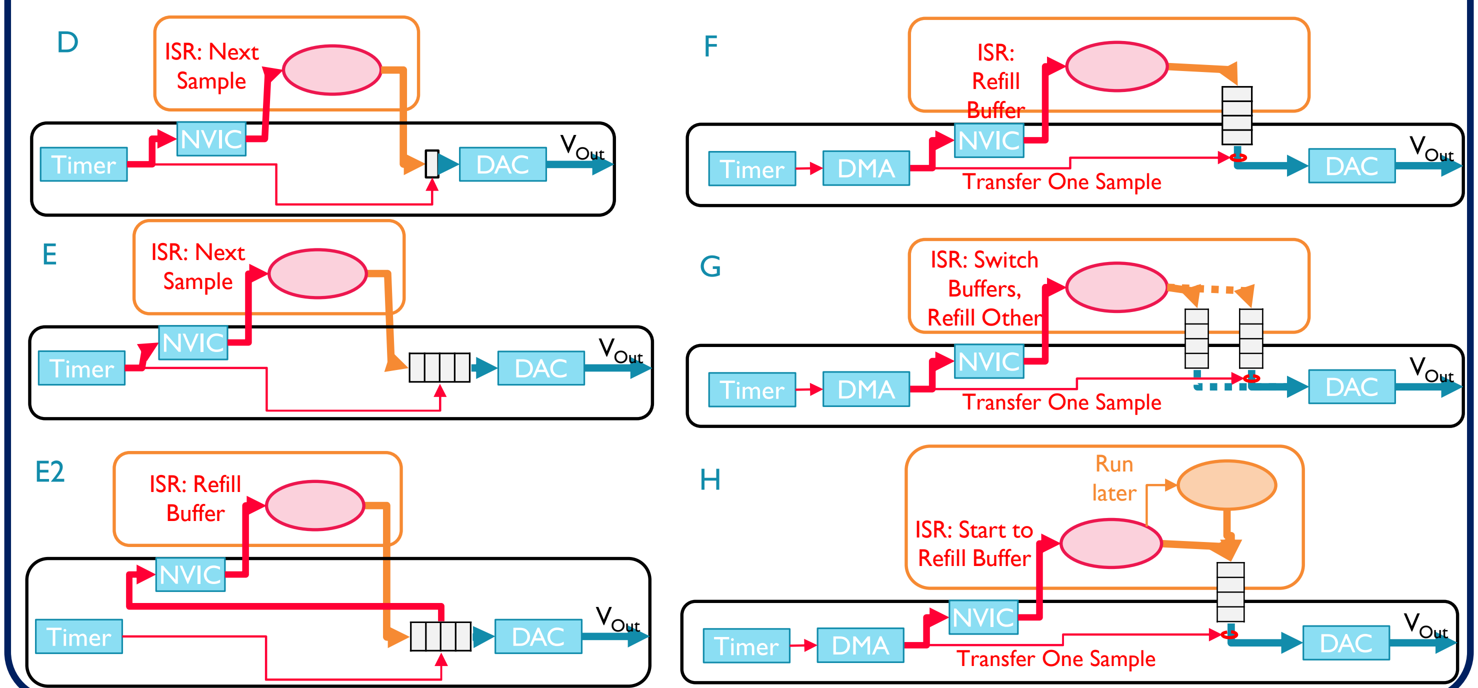


Software and Hardware Components in Design Evolution

***General Trend**: Move operations which need synchronization (update output, compute new value) from **software** to **hardware** to improve stability, performance*



***Asynchronous I/O:** Output is updated by hardware, so it is **decoupled** from (asynchronous with) **software execution**, so sloppy software timing affects output timing less*



Example Code from ESF

NC STATE UNIVERSITY

```
void Play_Tone(unsigned int period, unsigned int
num_cycles, unsigned wave_type) {
 unsigned step, out_data;

 while (num_cycles>0) {
  num_cycles--;
  for (step = 0; step < NUM_STEPS; step++) {
   switch (wave_type) {
    case SQUARE:
     if (step < NUM_STEPS/2)
      out_data = 0;
     else
      out_data = MAX_DAC_CODE;
     break;
    case RAMP:
     out_data = (step*MAX_DAC_CODE)/NUM_STEPS;
     break;
    case SINE:
     out_data = SineTable[step];
     break;
   default:
     break;
   }
   DAC0->DAT[0].DATH = DAC_DATH_DATA1(out_data >> 8);
   DAC0->DAT[0].DATL = DAC_DATL_DATA0(out_data);
   Delay_us(period/NUM_STEPS);
   }
  }
 }
```

NC STATE UNIVERSITY

Vulnerability to Timing Interference

A. Task busy-waits for constant time (blind), then writes to DAC

A2. OS runs task periodically, task writes to DAC

B. Task software poll/blocks on timer, then writes to DAC

C. Timer ISR writes data to DAC

D. Timer advances buffer data to DAC, Timer ISR writes next data to buffer

E. Timer advances buffer data to DAC. Timer ISR writes data to buffer

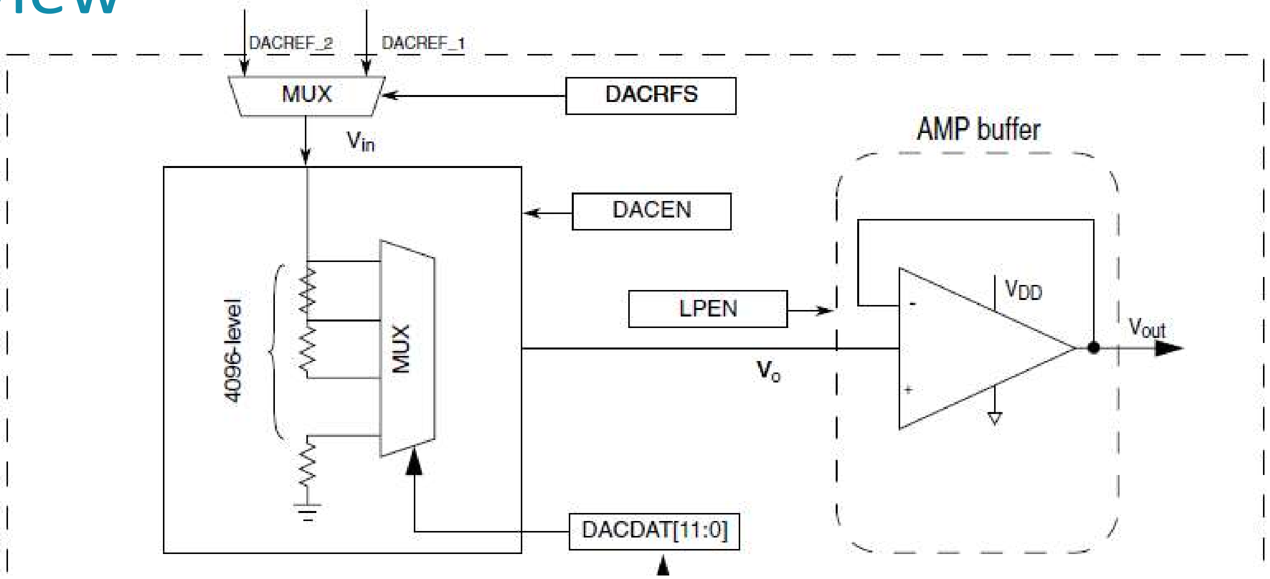
E2. Timer advances buffer data to DAC. Low/Empty ISR writes next batch of data to buffer

F. Timer triggers DMA data transfer, DMA ISR writes next batch of data to buffer

G. Timer triggers DMA transfer, DMA ISR switches buffers and writes next batch of data to previous buffer

H. Timer triggers DMA transfer, DMA ISR writes urgent data to buffer, triggers task. Task writes rest of data batch

DAC Overview



- Load DACDAT with 12-bit data N
- MUX selects a node from resistor divider network to create
  $V_o = (N+1)*V_{in}/2^{12}$
- V_o is buffered by output amplifier to create V_{out}
  - $V_o = V_{out}$ but V_o is high impedance - can't drive much of a load, so need to buffer it

DAC Registers

DAC memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
4003_F000	DAC Data Low Register (DAC0_DAT0L)	8	R/W	00h	30.4.1/531
4003_F001	DAC Data High Register (DAC0_DAT0H)	8	R/W	00h	30.4.2/532
4003_F002	DAC Data Low Register (DAC0_DAT1L)	8	R/W	00h	30.4.1/531
4003_F003	DAC Data High Register (DAC0_DAT1H)	8	R/W	00h	30.4.2/532
4003_F020	DAC Status Register (DAC0_SR)	8	R	02h	30.4.3/532
4003_F021	DAC Control Register (DAC0_C0)	8	R/W	00h	30.4.4/533
4003_F022	DAC Control Register 1 (DAC0_C1)	8	R/W	00h	30.4.5/534
4003_F023	DAC Control Register 2 (DAC0_C2)	8	R/W	0Fh	30.4.6/534

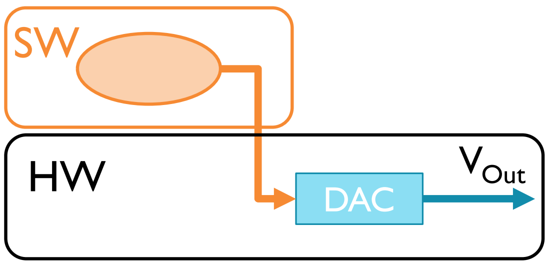
- This peripheral's registers are only eight bits long (legacy peripheral).

- DATA[11:0] stored in two registers
  - DATA0: Low byte [7:0] in DACx_DATnL
  - DATA1: High nibble [11:8] in DACx_DATnH

NC STATE UNIVERSITY

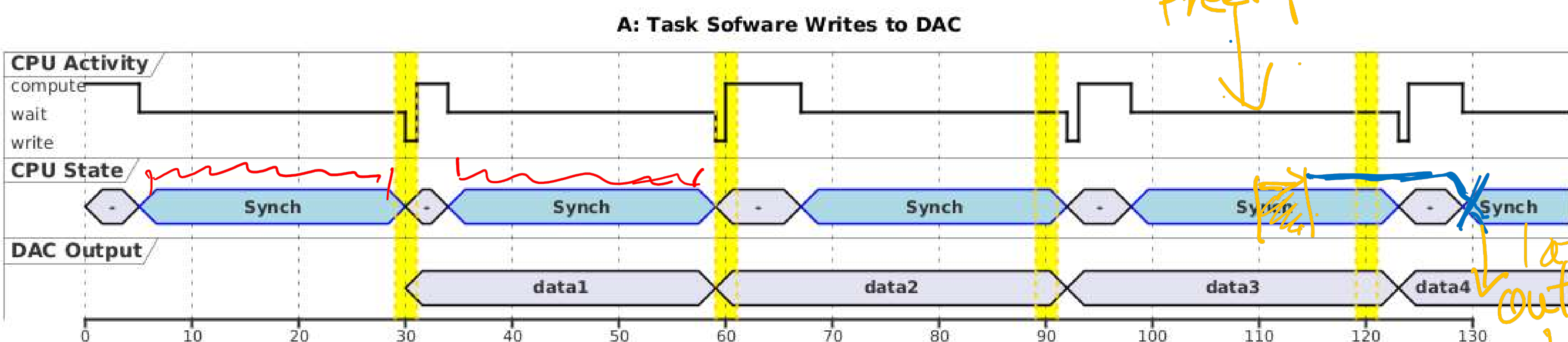
WAVEGEN STARTING POINT: DESIGN A

A. Simple Starter Code




```
while (1) {
    compute data
    // Synchronize: Wait until time for next sample
    for (t = T; t>0; t--); // busy wait loop creates delay
    // Position of following code implicitly schedules it
    write data to DAC
}
```

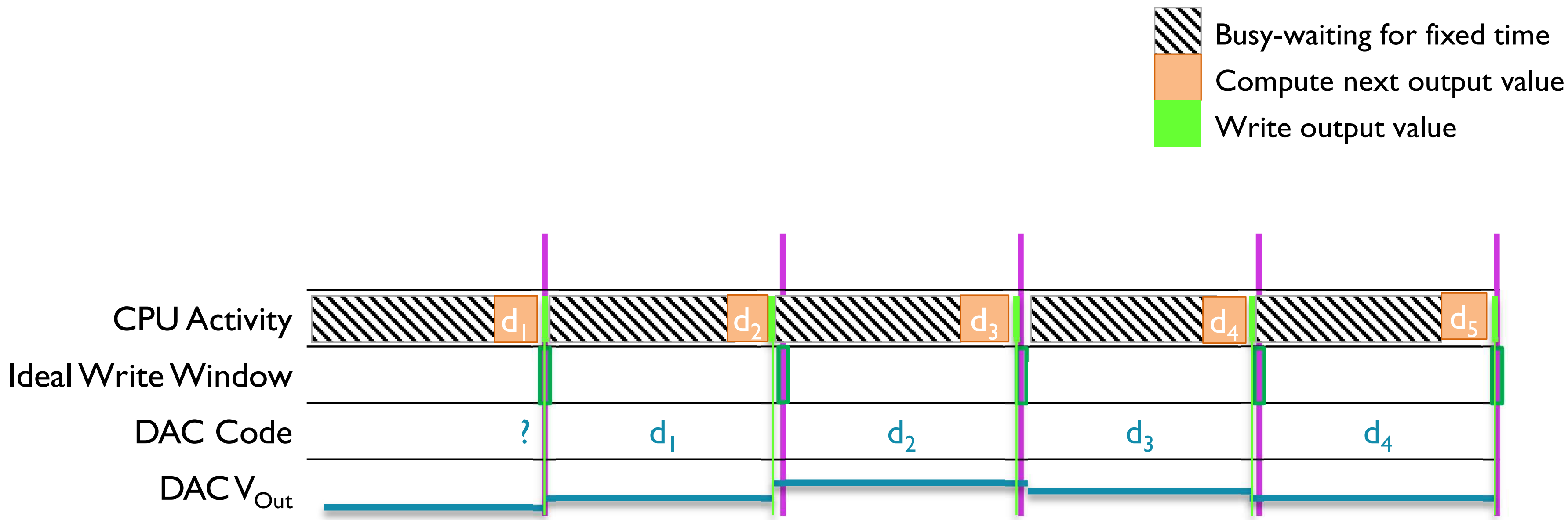
Preemption



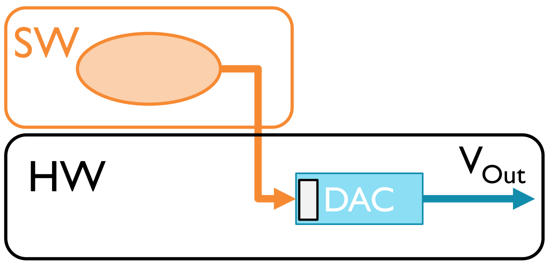
late output change

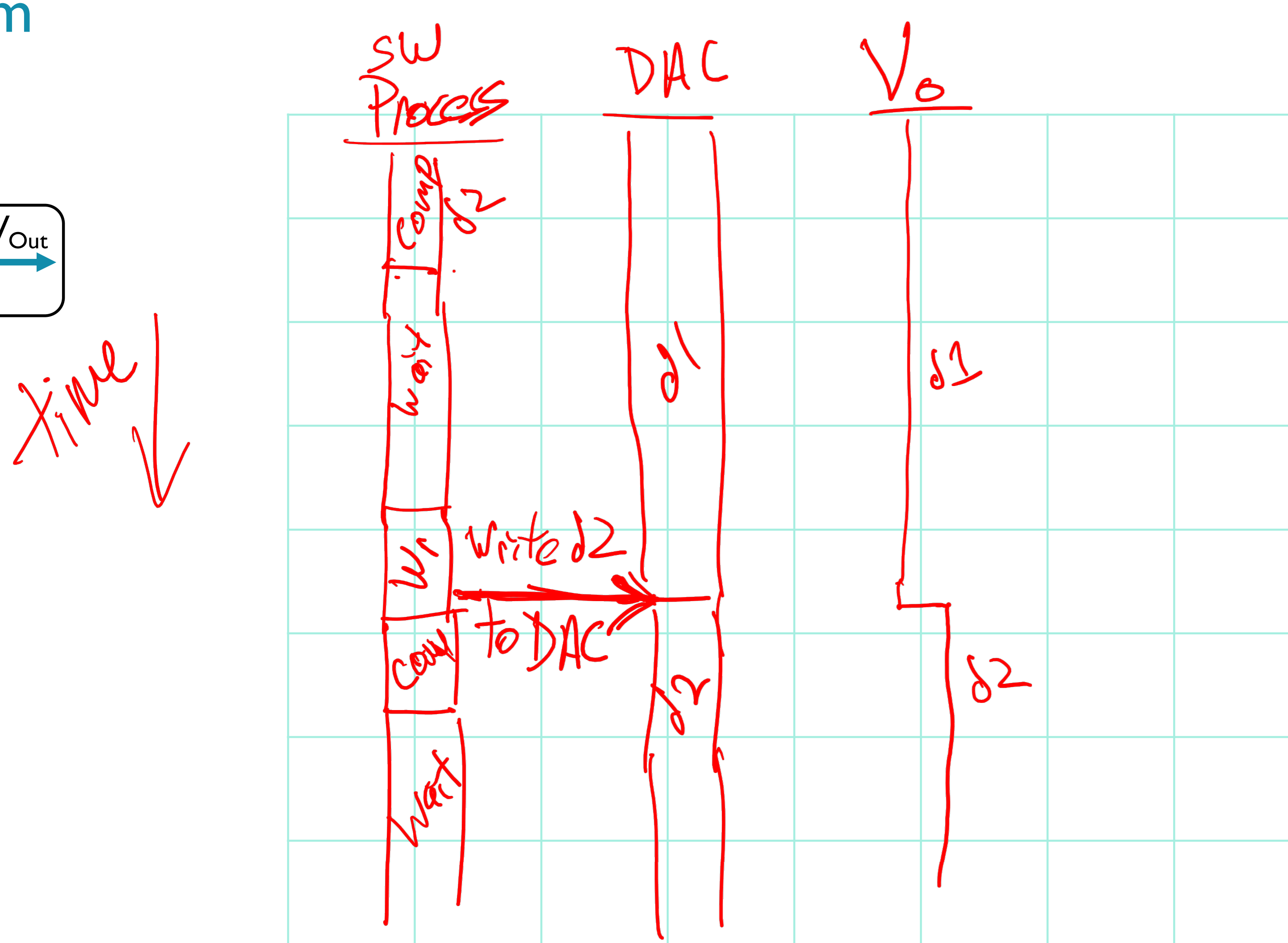
- Timing is unstable. Make T what value?
  - What if computing data takes variable time?
  - Vulnerable to interference by other handlers, processes on CPU
- Using busy-waiting to create time delay is greedy because it doesn't share CPU
- Synchronization and scheduling done completely in software

Unbuffered DAC with Busy-Wait Code



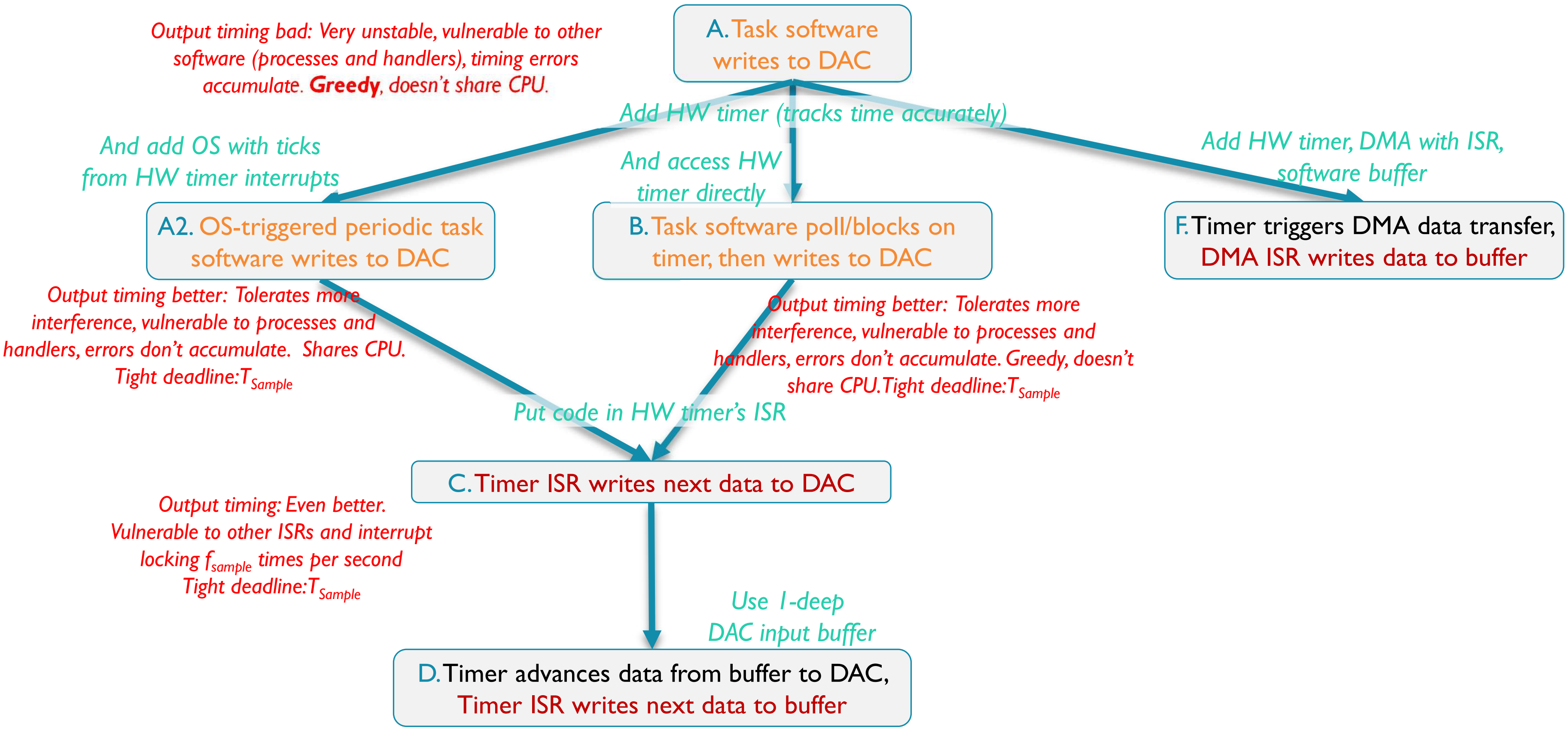
Sequence Diagram



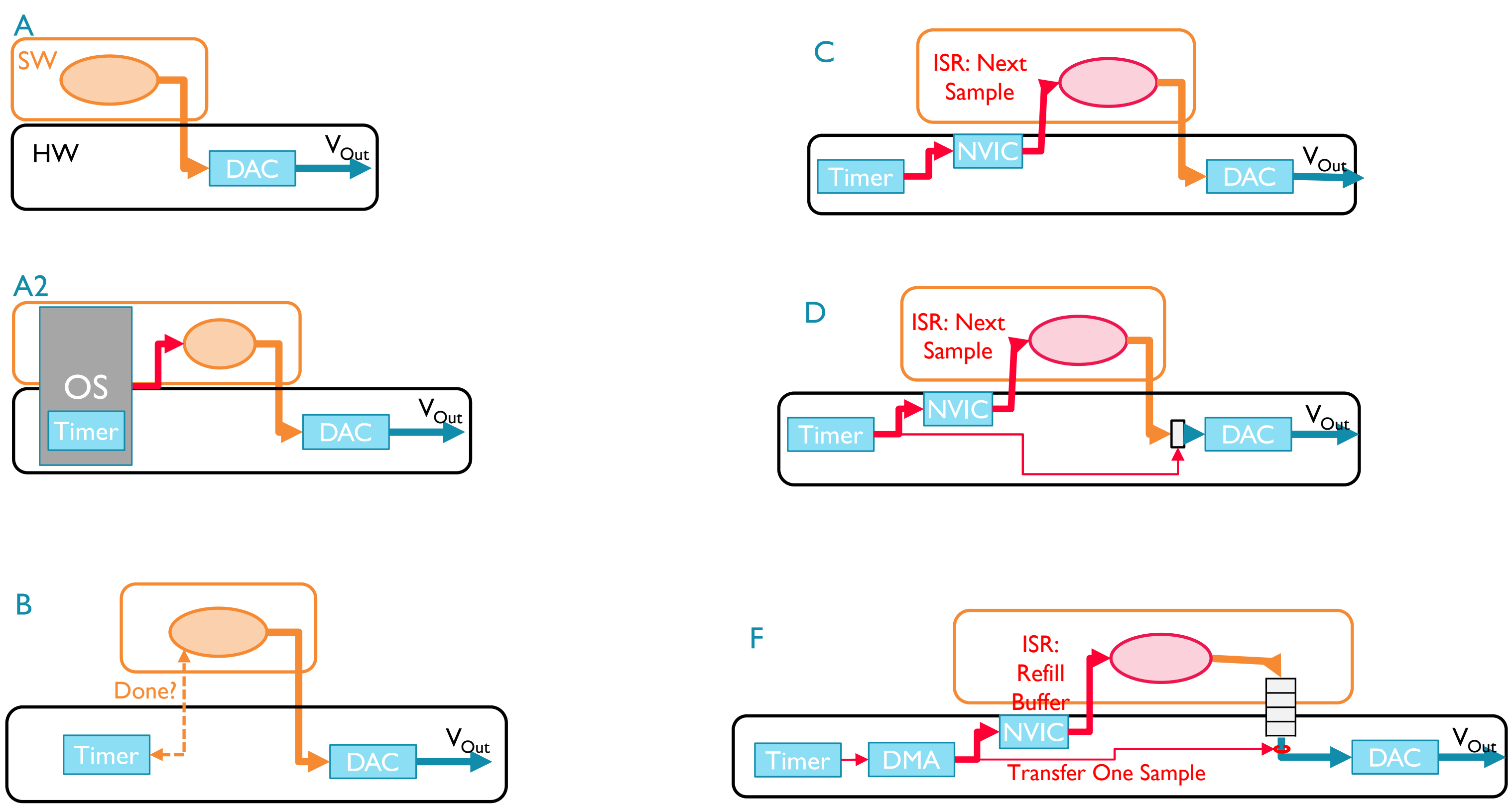


IMPROVING OUTPUT TIMING STABILITY

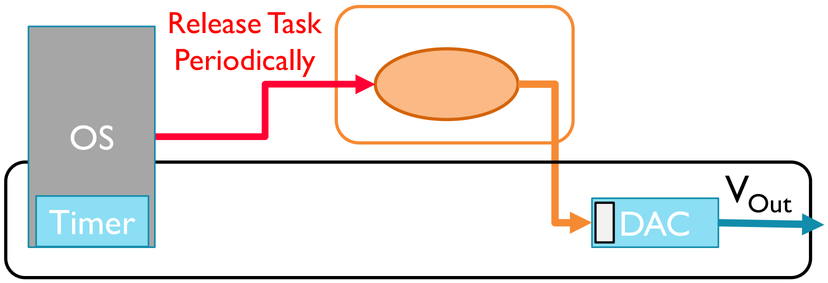
Detailed Overview of What and Why



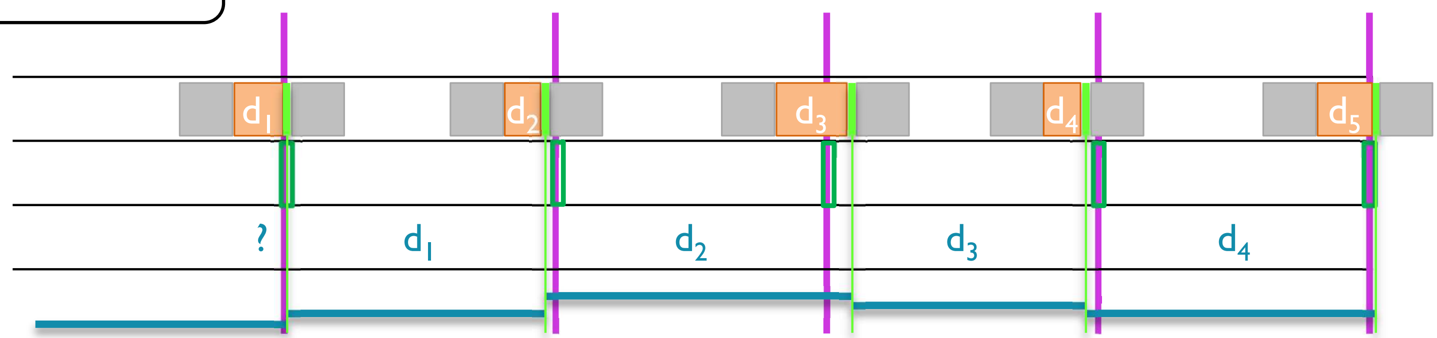
Software and Hardware Components



A2. Scheduler Releases WaveGen Periodically

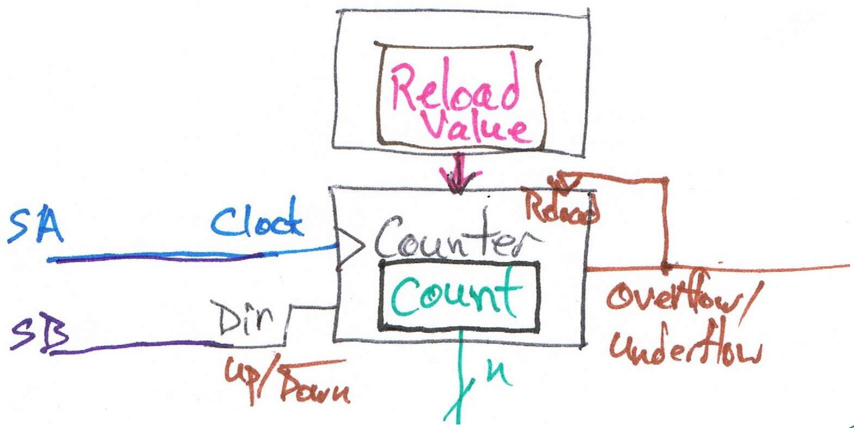


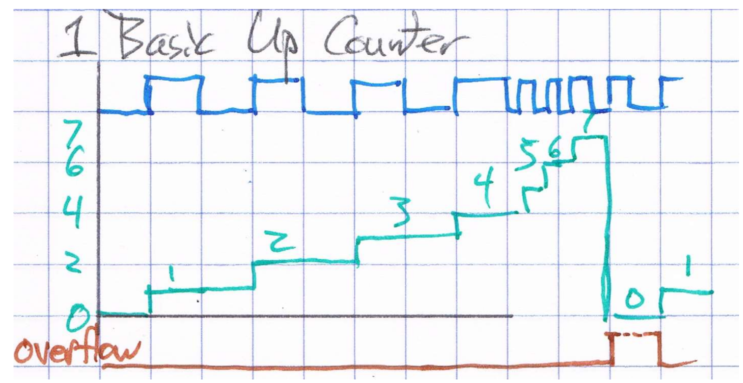

```
void Task_WaveGen (void) {
    compute data
    write data to DAC
}
```

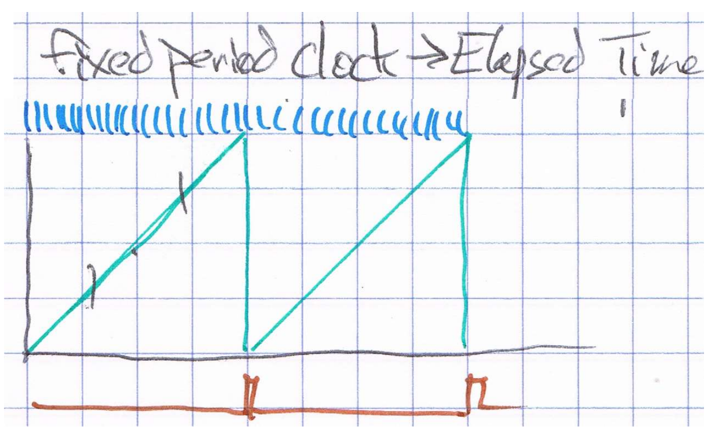


- DAC update is on time if …
  - Task_WaveGen starts on time (tasks/ISRs finish early enough for scheduler to run it), and
  - Task_WaveGen not preempted by other processing
- DAC update is delayed if other processing (tasks, ISRs) cause timing interference:
  - Task_WaveGen starts late if tasks/ISRs finish too late (delaying scheduler), so DAC is updated late
  - Task_WaveGen updates DAC late if preempted by higher-priority software processes (e.g. ISRs)

B. Software Polls Hardware Timer

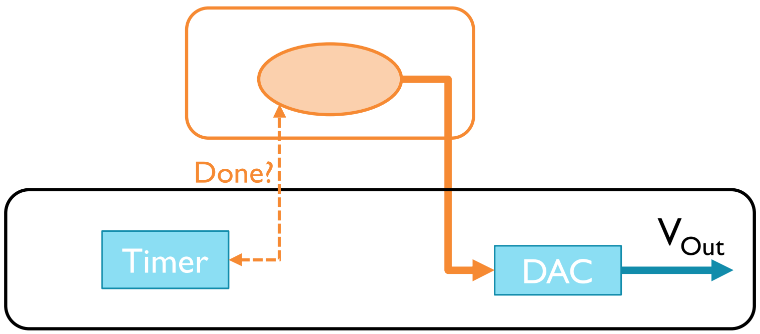







```
TNext = TSample
while (1) {
     compute data
     // Synchronize: Wait until time for next sample
     while (TimerValue < TNext); // busy wait polling loop
     // Position of following code implicitly schedules it
     write data to DAC
     TNext += TSample
}
```

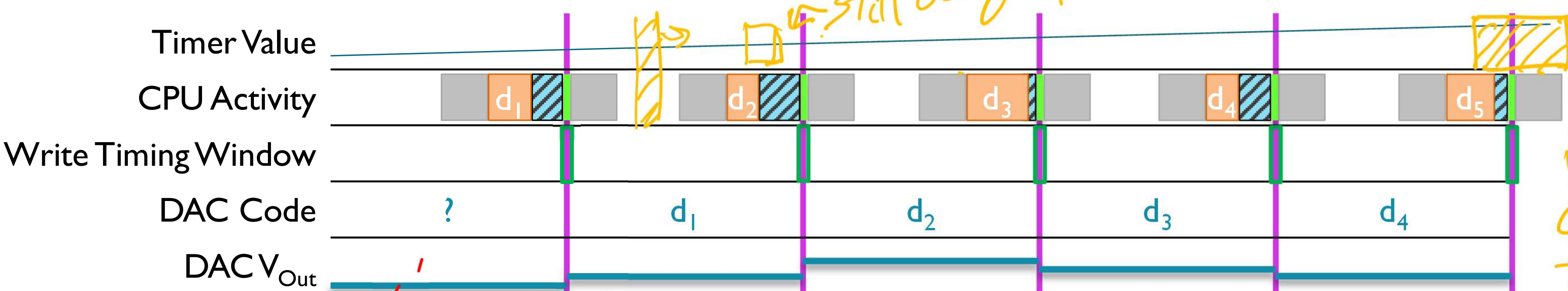
B. Software Polls Hardware Timer




```
TNext = TSample
while (1) {
    compute data
    // Synchronize: Wait until time for next sample
    while (TimerValue < TNext); // busy wait polling loop
    // Position of following code implicitly schedules it
    write data to DAC
    TNext += TSample
}
```

sched

still doing sync, still finish on time

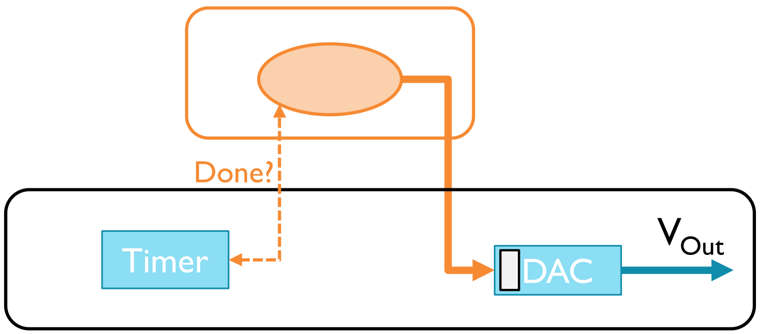


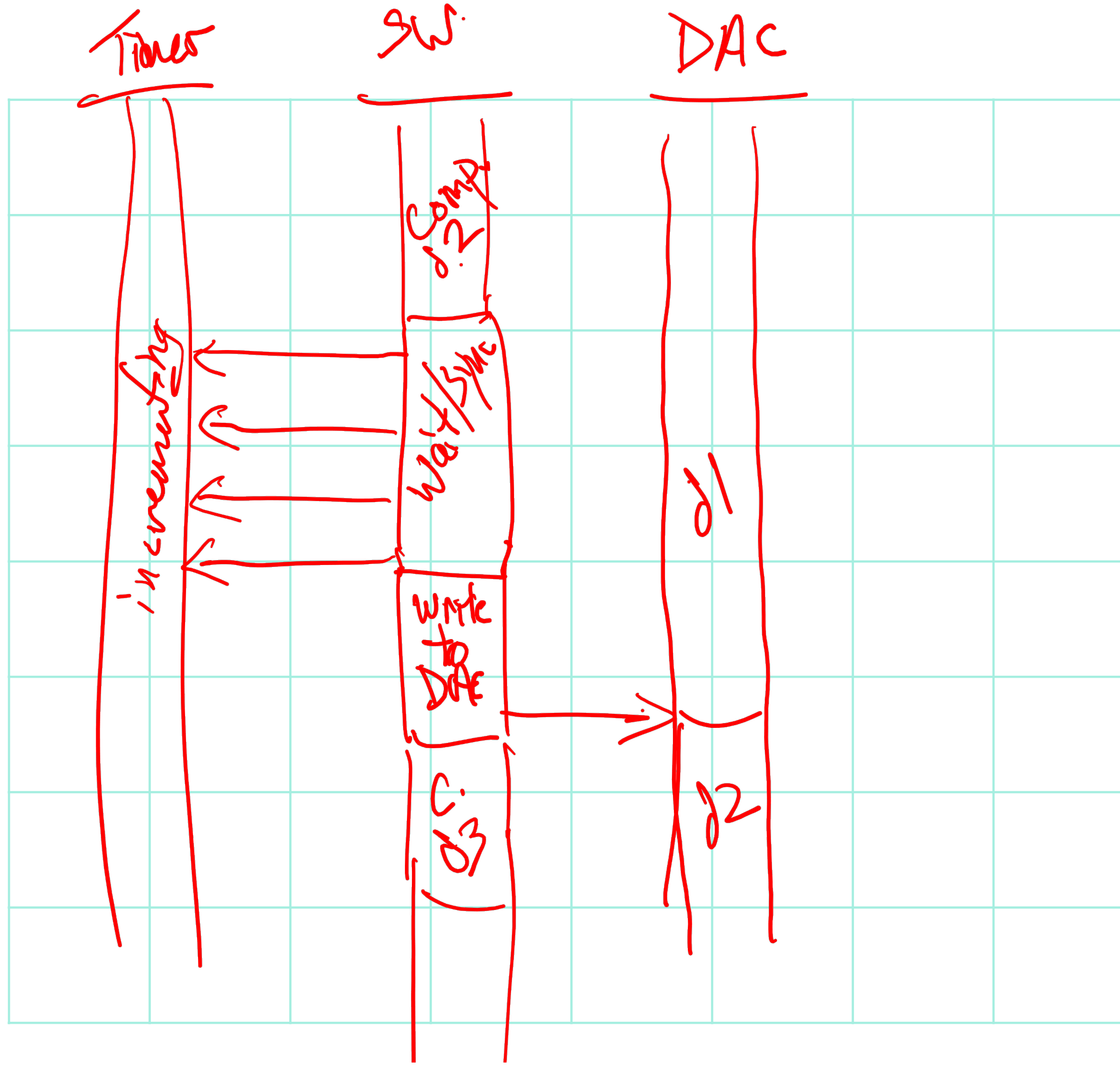
output too late

timing fail

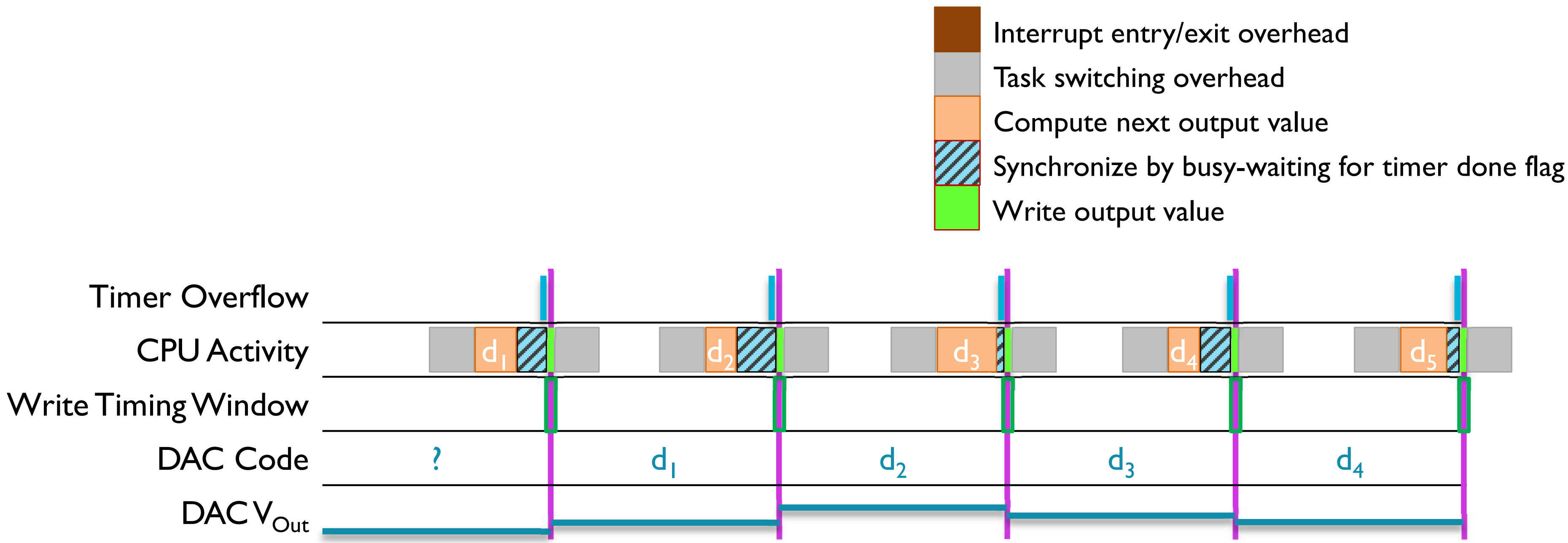
- Add hardware timer/counter peripheral
  - Binary counter (e.g. 16, 32 bits) tracks elapsed time by counting clock pulses. SW can read (e.g. TimerValue)
  - Increments periodically, regardless of SW activity
- Synchronization loop tolerates some timing interference
  - How much? Class discussion activity…
- Synch performed in SW with HW help
- Scheduling performed in SW

Sequence Diagram

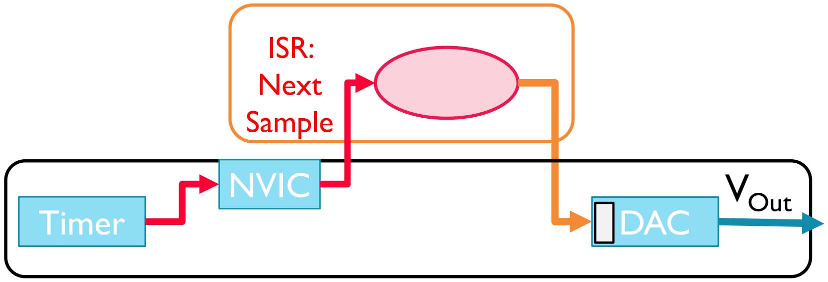




Unbuffered DAC with Timer Polling



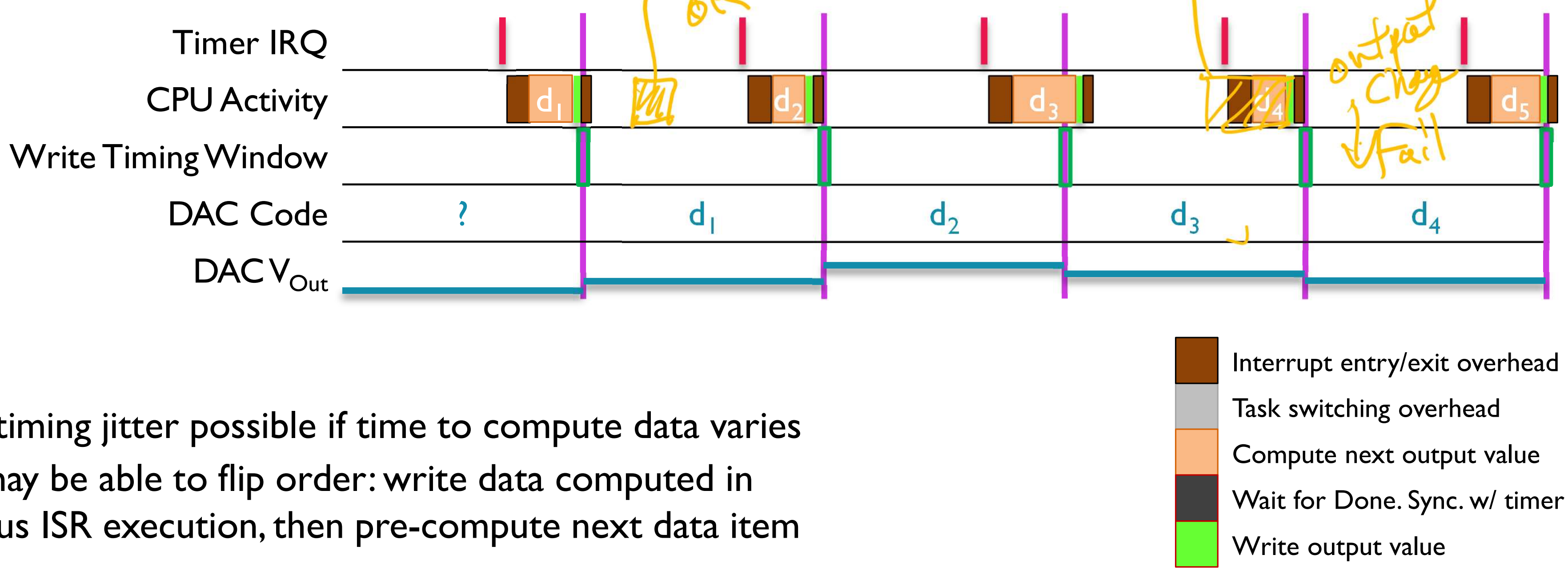
C. Hardware Timer Periodically Triggers Interrupt for DAC Write




```
Interrupt Handler {
  compute data
  write data to DAC
}
```

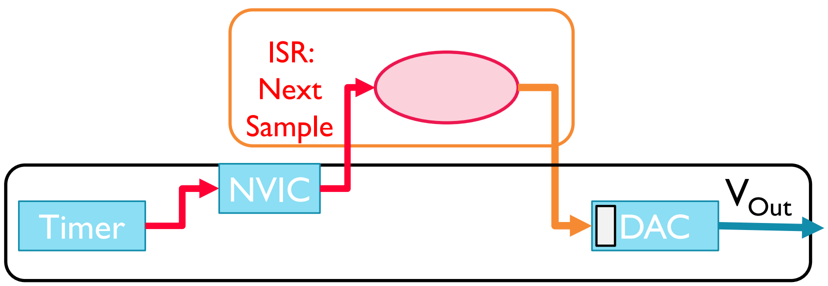
Interference OK

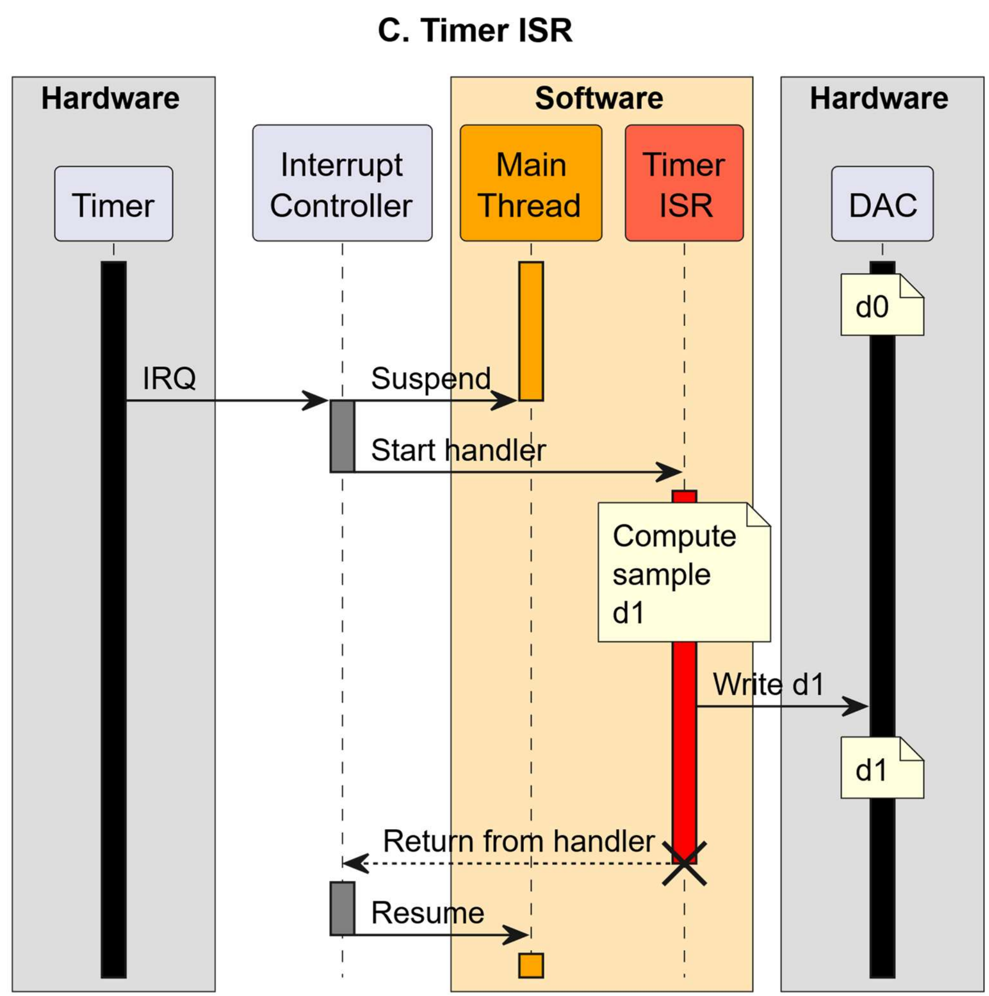
Higher Pri ISR Interference not OK



- Some timing jitter possible if time to compute data varies
- If so, may be able to flip order: write data computed in previous ISR execution, then pre-compute next data item

Sequence Diagram





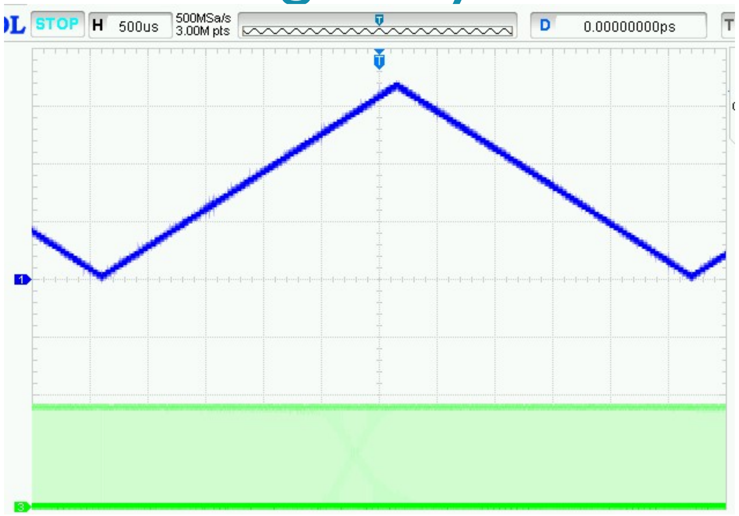
Example Source Code from ESF: Listings 7.7, 7.8

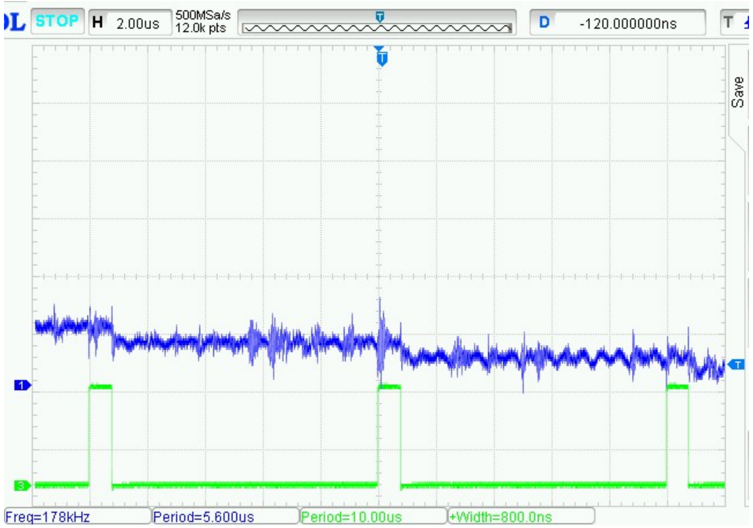
```
void Init_TPM(void)
{
   // Turn on clock to TPM
   SIM->SCGC6 |= SIM_SCGC6_TPM0_MASK;
   // Set clock source for tpm
   SIM->SOPT2 |= (SIM_SOPT2_TPMSRC(1) | SIM_SOPT2_PLLFLLSEL_MASK);
   // Load the counter and mod, given prescaler of 32
   TPM0->MOD = (F_TPM_CLOCK/(F_TPM_OVFLW*32))-1;
   // Set TPM to divide by 32 prescaler, enable counting (CMOD) and interrupts
   TPM0->SC = TPM_SC_CMOD(1) | TPM_SC_PS(5) | TPM_SC_TOIE_MASK;
   // Enable interrupts in NVIC
   NVIC_SetPriority(TPM0_IRQn, 3);
   NVIC_ClearPendingIRQ(TPM0_IRQn);
   NVIC_EnableIRQ(TPM0_IRQn);
}
```

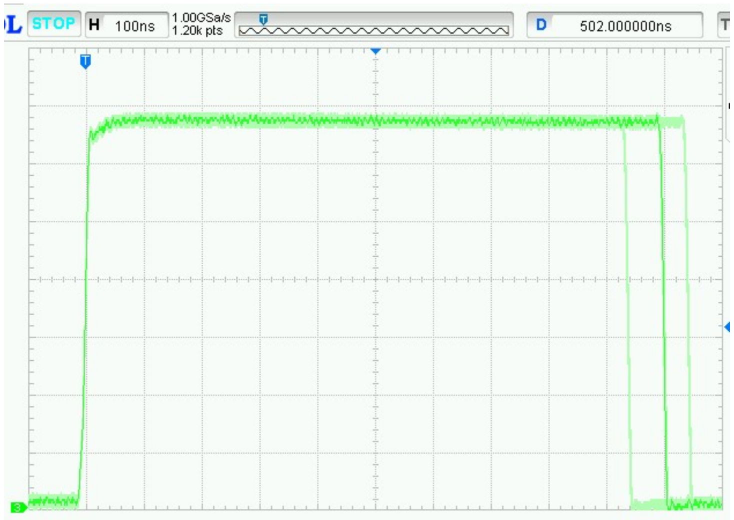
```
void TPM0_IRQHandler() {
   static int change=STEP_SIZE;
   static uint16_t out_data=0;

   FPTD->PSOR = MASK(BLUE_LED_POS); // Debug signal: Entering ISR
   TPM0->SC |= TPM_SC_TOIE_MASK;    // reset overflow flag

   // Do ISR work
   out_data += change;
   if (out_data < STEP_SIZE) {
             change = STEP_SIZE;
   } else if (out_data >= DAC_RESOLUTION-STEP_SIZE) {
             change = -STEP_SIZE;
   }
   DAC0->DAT[0].DATH = DAC_DATH_DATA1(out_data >> 8);
   DAC0->DAT[0].DATL = DAC_DATL_DATA0(out_data);
   FPTD->PCOR = MASK(BLUE_LED_POS); // Debug signal: Exiting ISR
}
```

- ESF code generates each waveform sample in ISR (IRQ Handler)

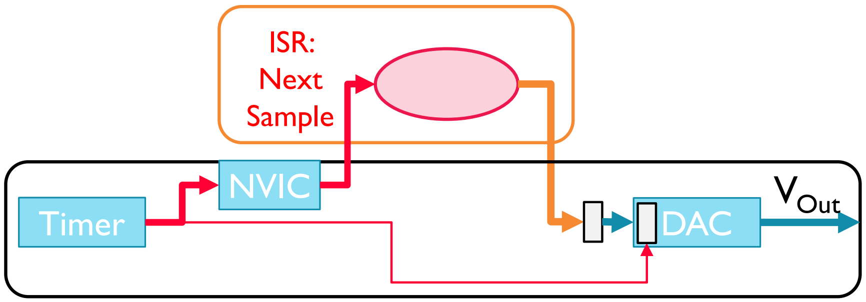
C. Timing Analysis





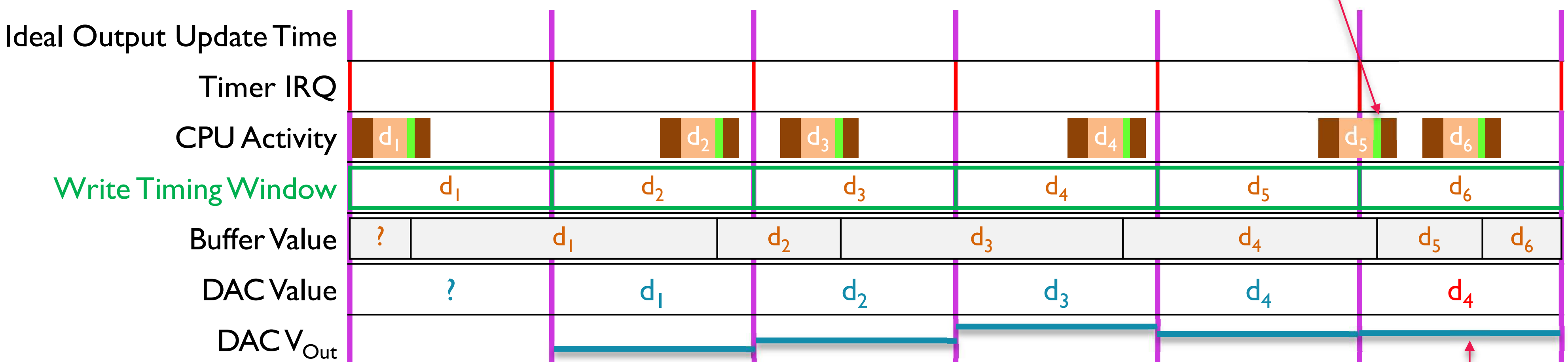


D. Add 1-element Input Data Buffer for DAC




```
Interrupt Handler {
     compute data
     write data to DAC
}
```

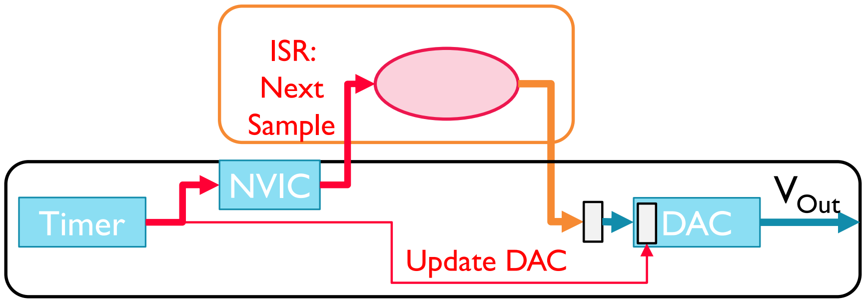
Error: d_5 not updated within its timing window



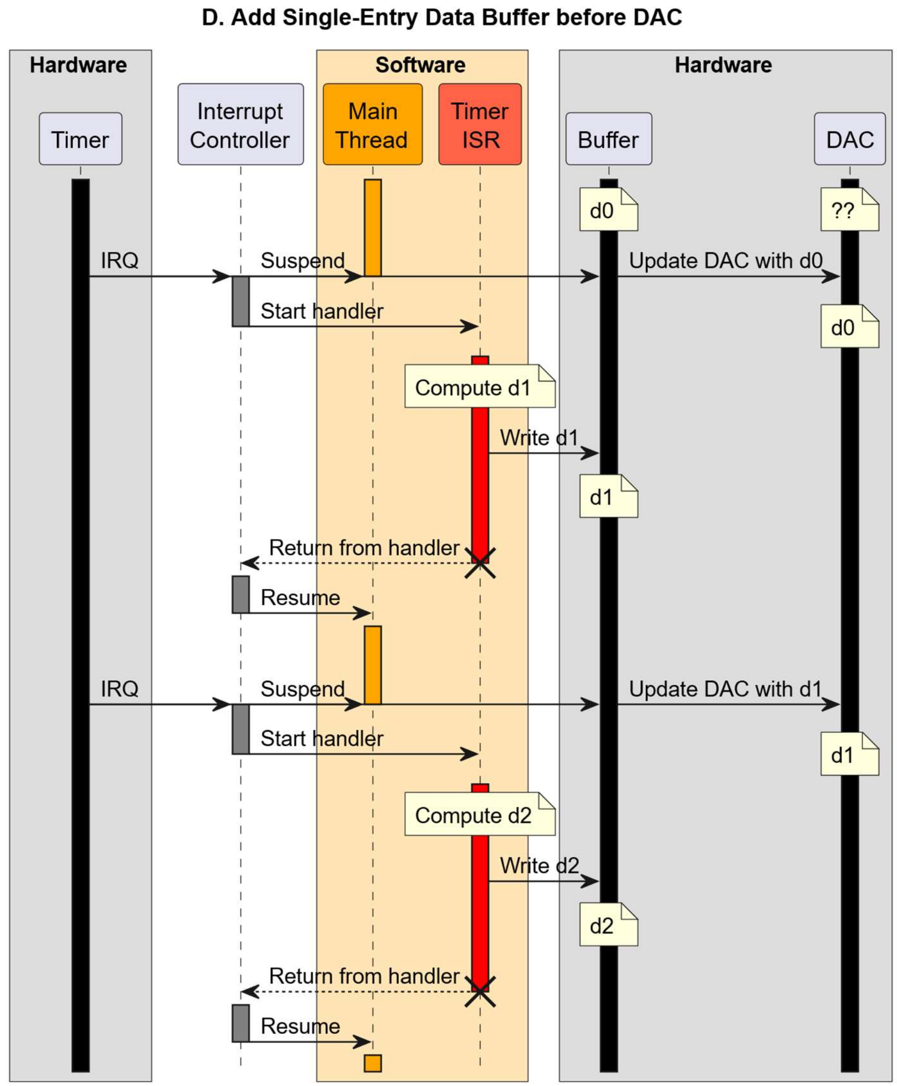
Result: d_4 is repeated

- Timer also triggers DAC to update input data register from external data buffer
- Write timing window is now up to T_{sample} (50 µs) before ideal output update time

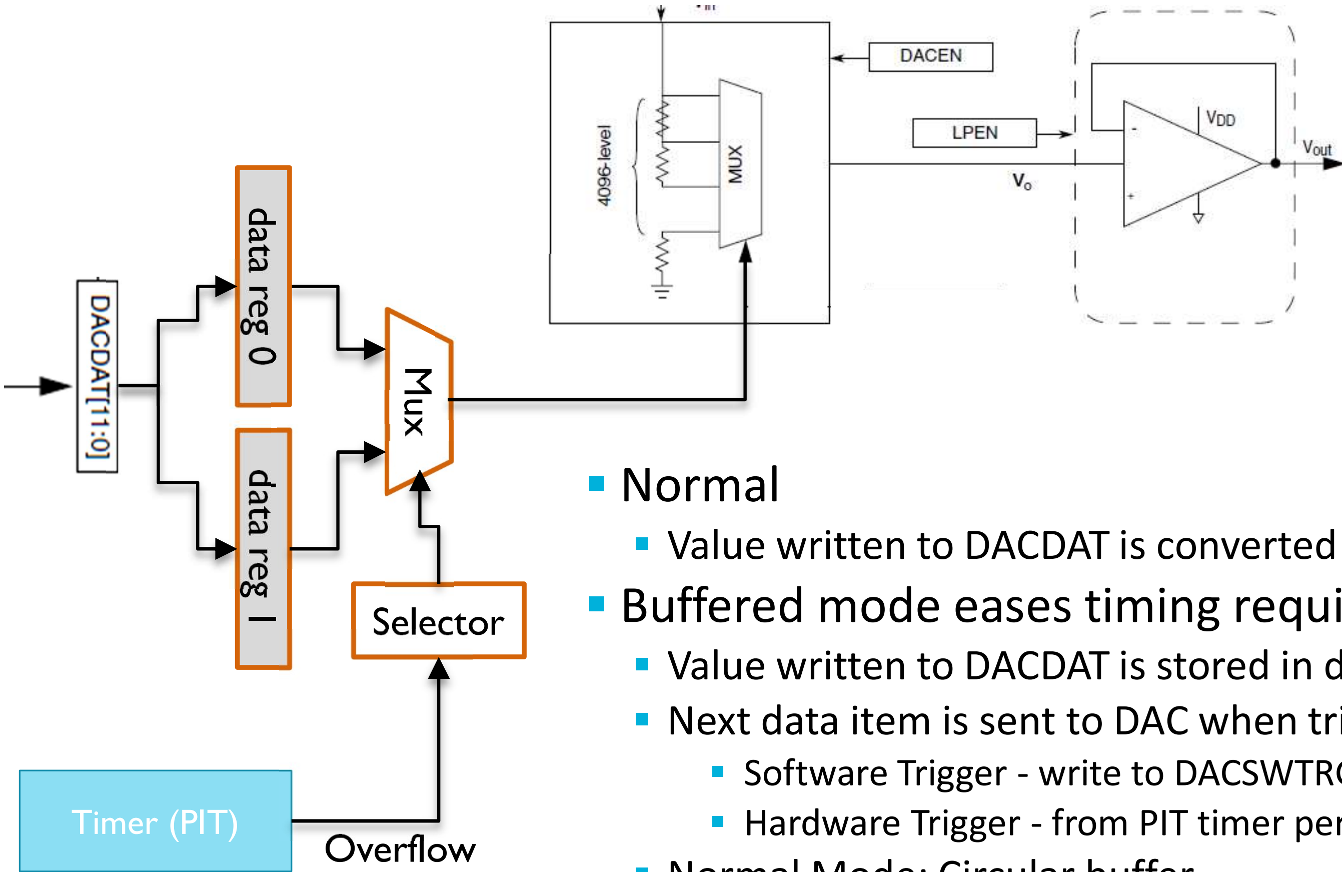
Sequence Diagram



- ISR can start slightly late and still update buffer in time
  - Slack time: $T_{sample} - T_{compute} - T_{to\ be\ determined}$

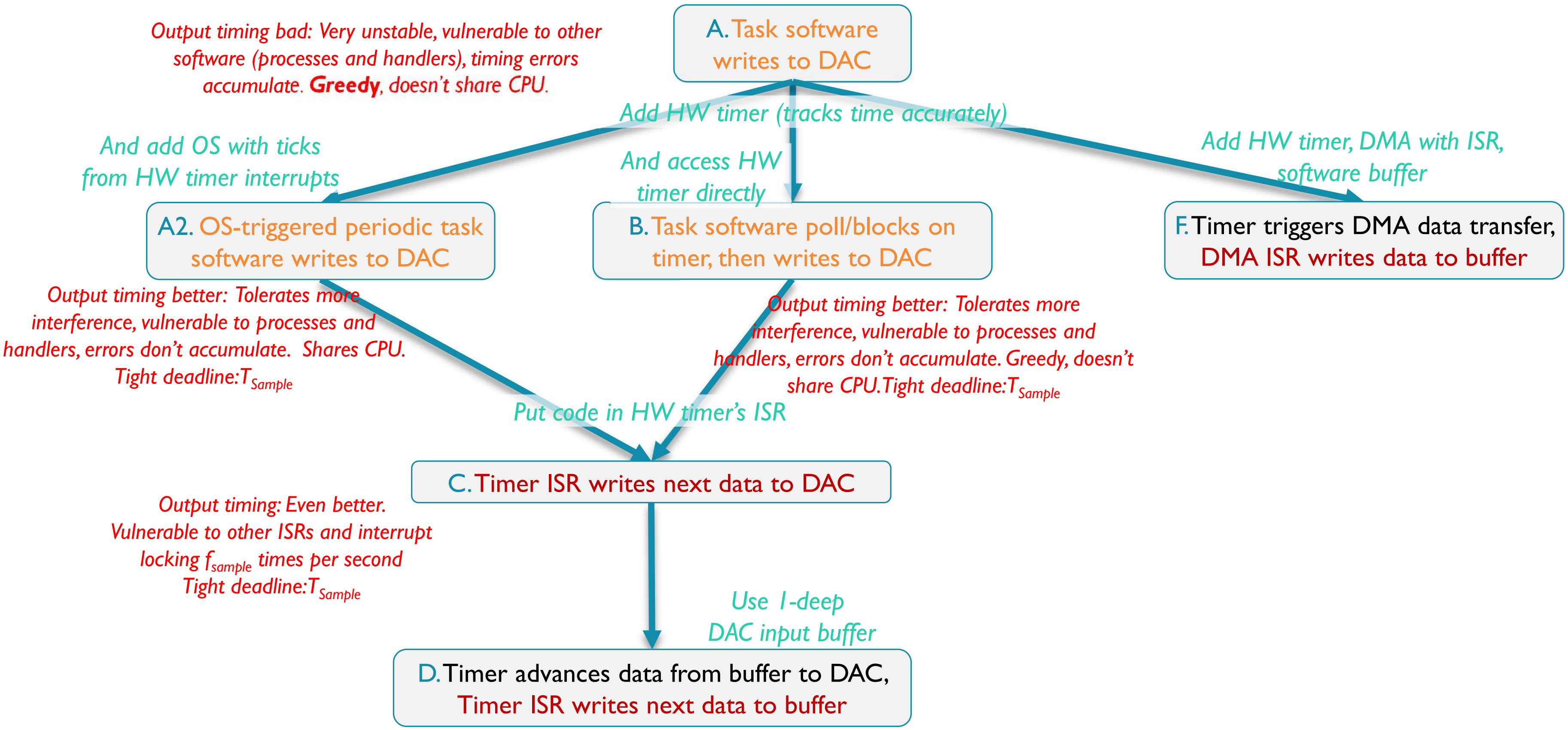


DAC Operating Modes

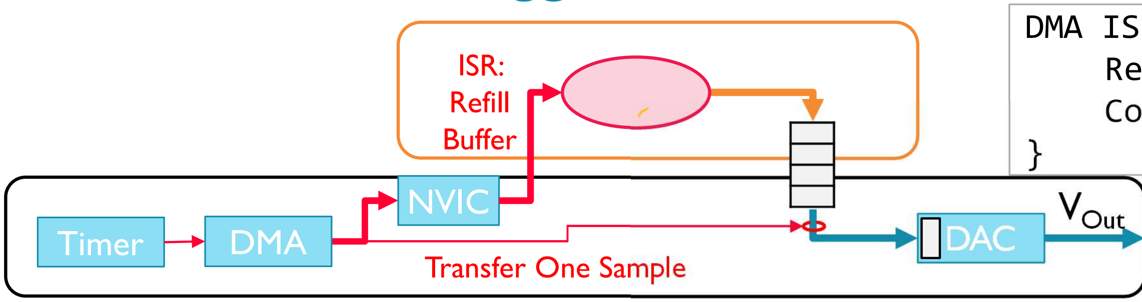


- Normal
  - Value written to DACDAT is converted to voltage immediately
- Buffered mode eases timing requirements
  - Value written to DACDAT is stored in data buffer for later conversion
  - Next data item is sent to DAC when triggered
    - Software Trigger - write to DACSWTRG field in DACx_C0
    - Hardware Trigger - from PIT timer peripheral
  - Normal Mode: Circular buffer
  - One-time Scan Mode: Pointer advances, stops at end of buffer
  - Status flags in DACx_SR

Detailed Overview of What and Why

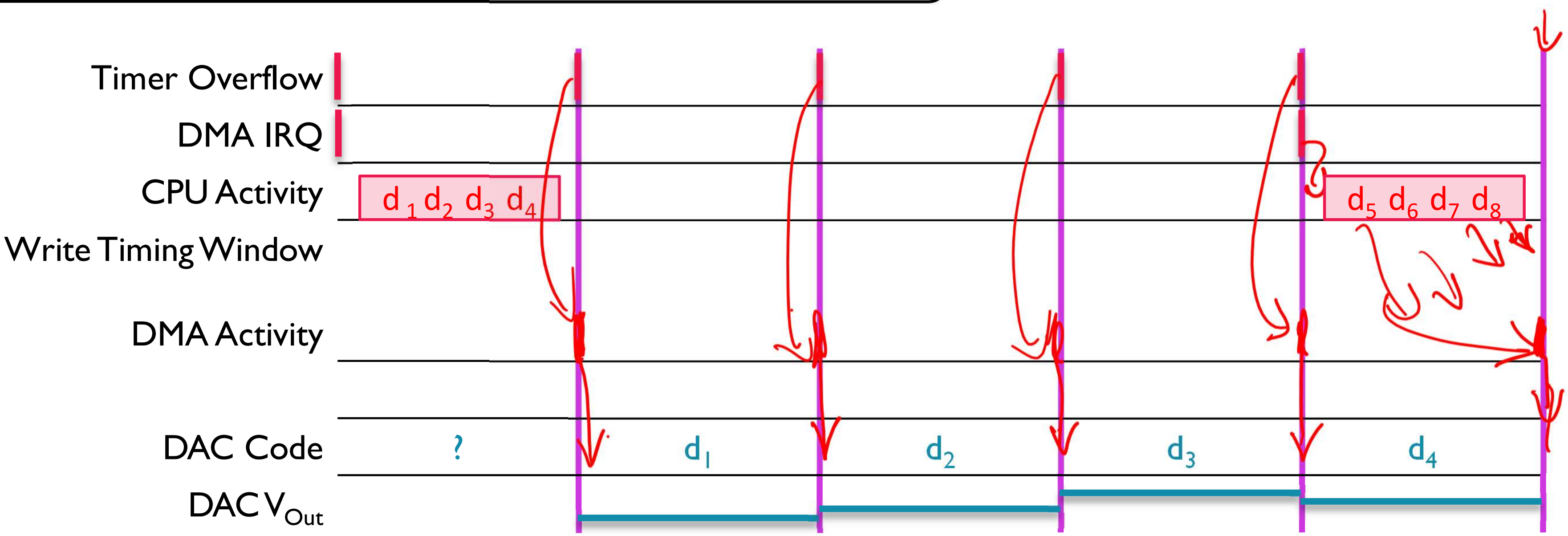


F. Use Timer-Triggered DMA to Transfer Data

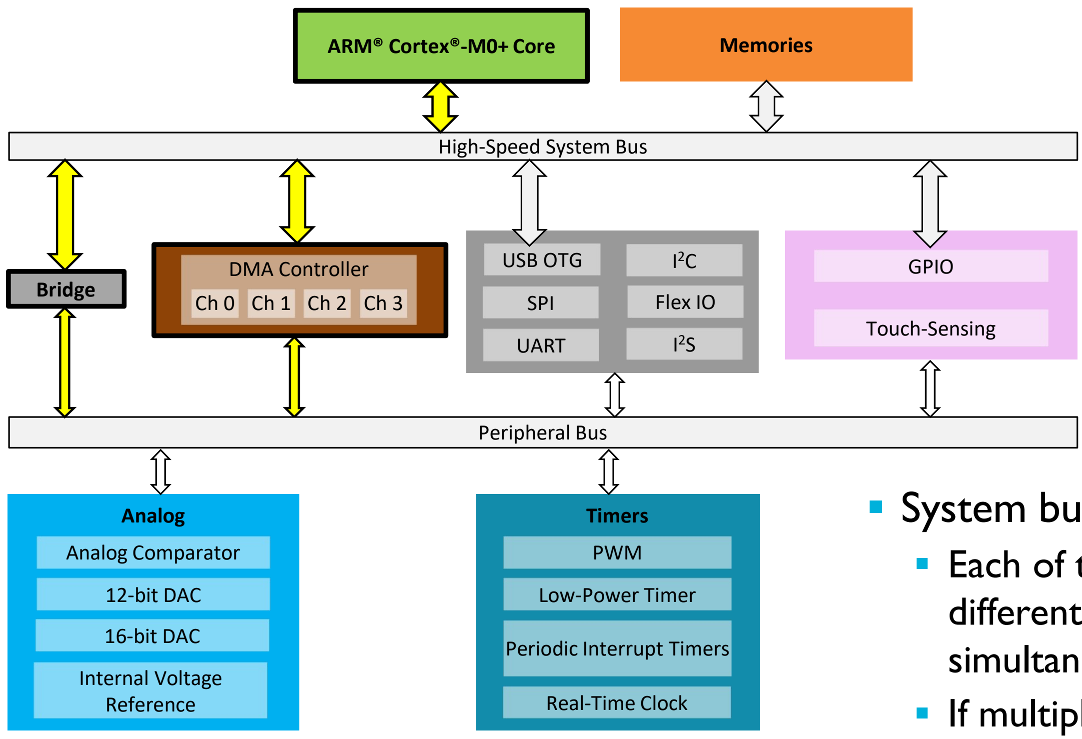


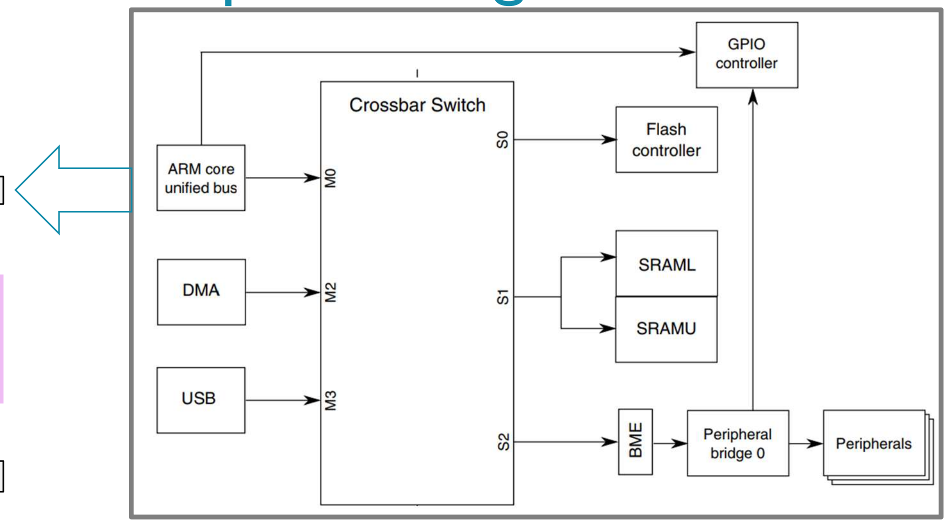

```
DMA ISR {
     Restart DMA channel
     Compute new data and refill buffer
}
```

ISR must save first new data value d_5 before next sample time



DMA Can Read and Write Memory and Peripheral Registers

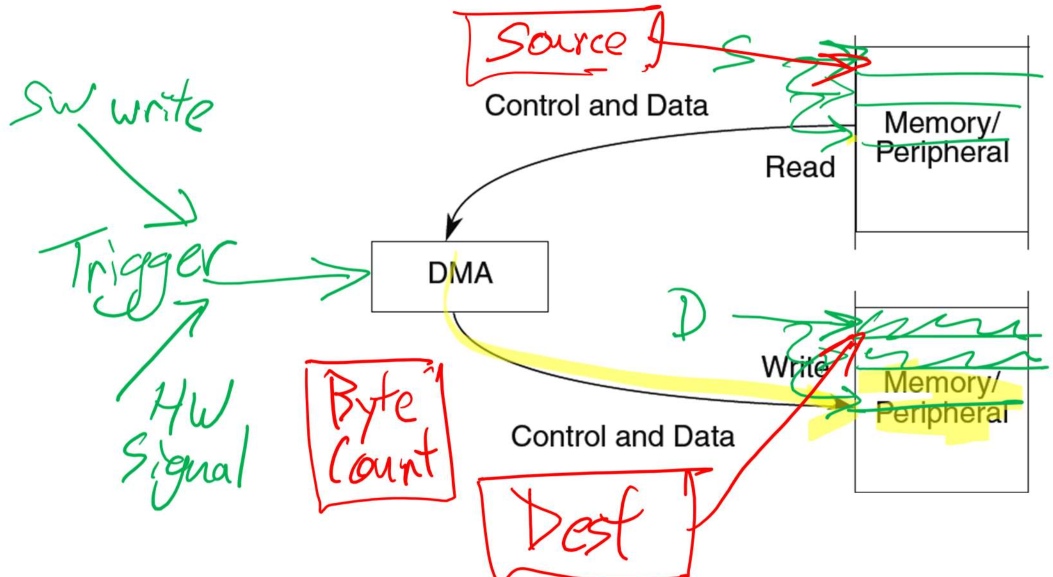




- System bus is a **crossbar switch**: three three-way switches
  - Each of three masters (CPU core, DMA, USB) can access a different device (flash memory, SRAM, peripheral bus) simultaneously
  - If multiple masters try to access same device, crossbar arbitrates (decides order for accesses)
- KL25Z Reference Manual
  - Ch. 20: Crossbar Switch Lite, Ch. 3 Section 4.6: Crossbar Configuration
  - Ch. 21: Peripheral Bridge

Basic Concepts

- Hardware which reads data from a source and writes it to a destination
  - Can copy data quickly
  - Can eliminate ISRs which just copy data (e.g. move ADC results into buffer)
- Example: Data copy
  - Source: source memory buffer
  - Destination: destination memory buffer
  - Trigger: software command
- Various configurable options
  - Number of data items to copy
  - Source and destination addresses can be fixed or change (e.g. increment, decrement)
  - Size of data item (1, 2, 4 bytes)
  - When transfer starts

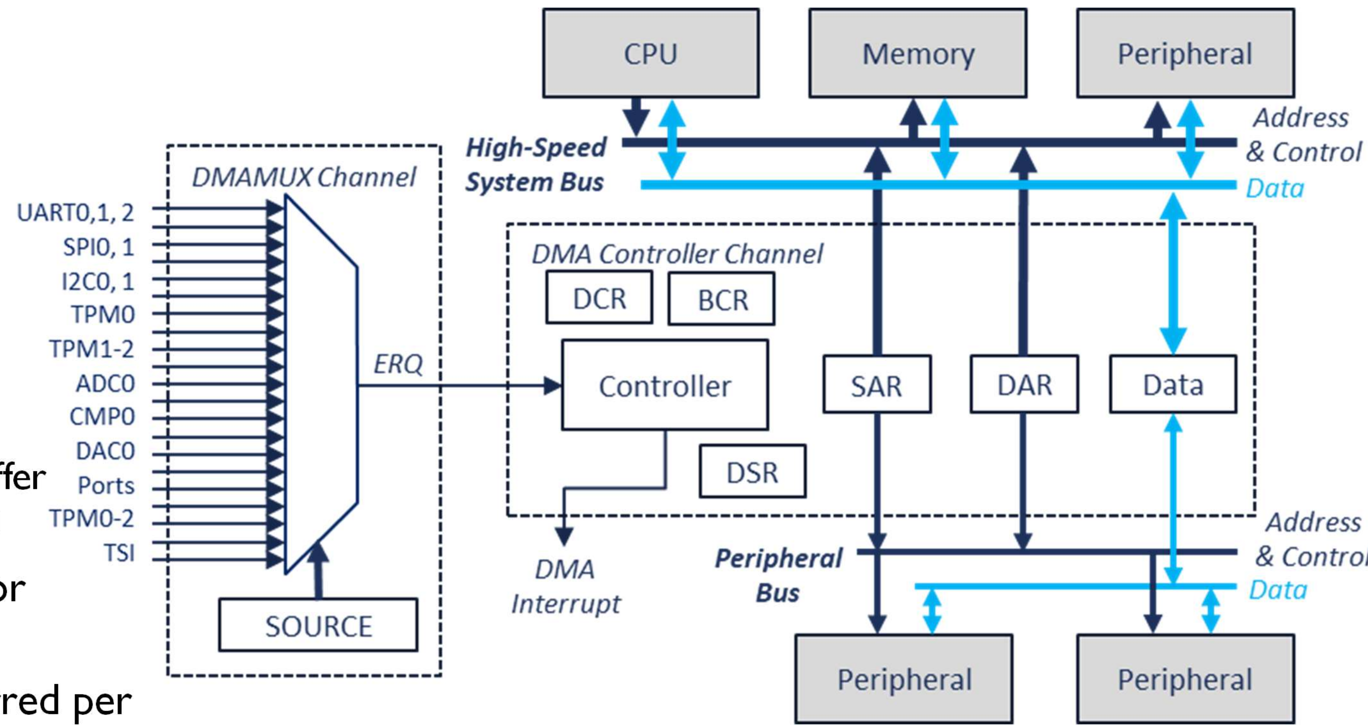


- Operation
  - Initialization: Configure controller
  - Transfer: Data is copied
  - Termination: Channel indicates transfer has completed

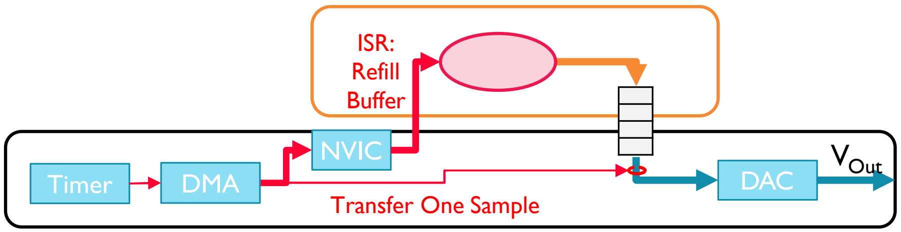
S	D	Op
+	+	Copy
•	+	Fill Buffer w/ Input Data
+	•	Send out Buffer Data

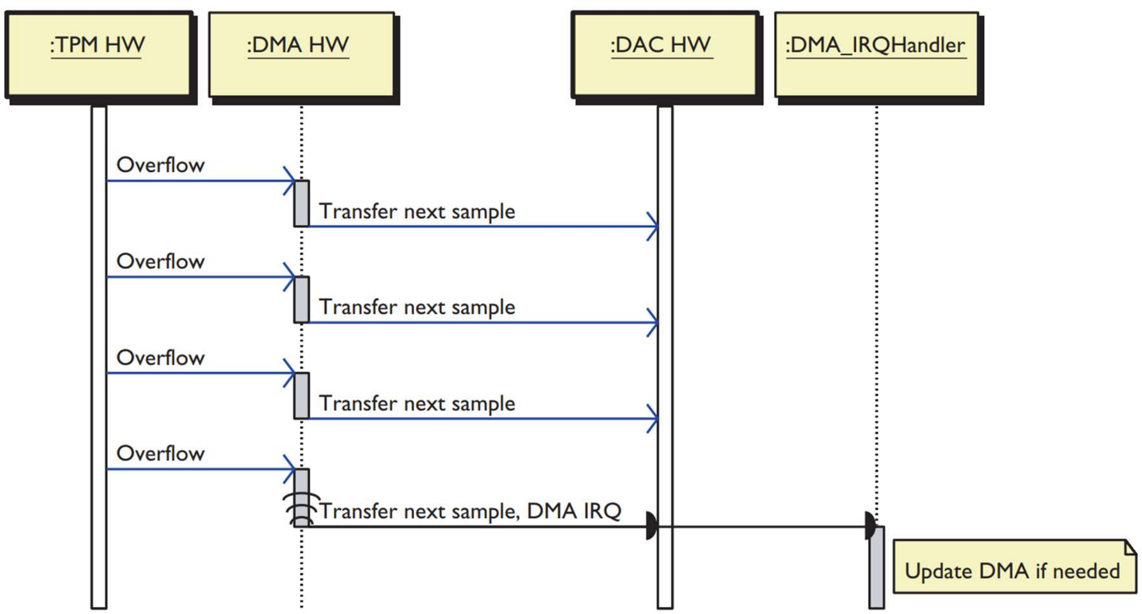
DMA Controller Details

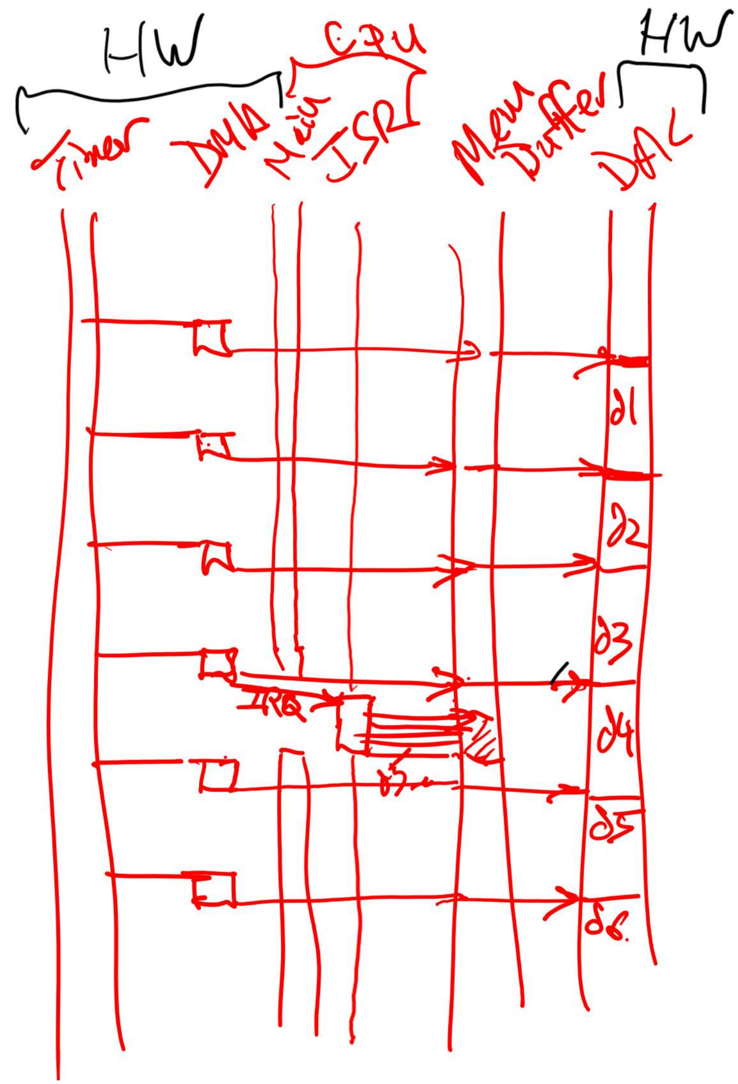
- 4 independent channels
  - Channel 0 has highest priority
- 8-, 16- or 32-bit transfers, data size can differ between source and destination
- Modulo addressability
  - Pointer wraps around at end of buffer
  - Enables ring buffers with size of 2^N
- Can trigger with hardware signal or software
- How many data items are transferred per trigger?
  - One: "Cycle stealing"
  - All items: grabs bus
- Hardware acknowledge/done signal



Sequence Diagram





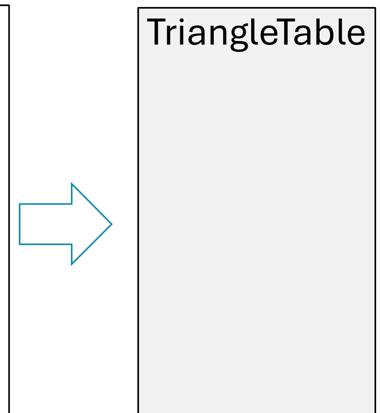


Example Source Code from ESF: Listings 9.4, 9.5

- ESF code initializes buffer (TriangleTable), then replays buffer contents repeatedly

```
void Play_Tone_with_DMA(void) {
	Init_RGB_LEDs();
	Control_RGB_LEDs(0,0,0);
	Init_DAC();
	Init_TriangleTable();
	Init_DMA_For_Playback(TriangleTable, NUM_STEPS);
	Init_TPM(10);
	Start_TPM();
	Start_DMA_Playback();
	while (1)

}
```

```
#define MAX_DAC_CODE (4095)
#define NUM_STEPS (512)

uint16_t TriangleTable[NUM_STEPS];

void Init_TriangleTable(void) {
	unsigned n, sample;

	for (n=0; n<NUM_STEPS/2; n++) {
		sample = (n*(MAX_DAC_CODE+1)/(NUM_STEPS/2));
		TriangleTable[n] = sample; // Fill in from front
		TriangleTable[NUM_STEPS-1-n] = sample; // Fill in from back
	}
}
```



Example Source Code from ESF: Listings 9.6

```
uint16_t * Reload_DMA_Source=0;
uint32_t Reload_DMA_Byte_Count 1024
```

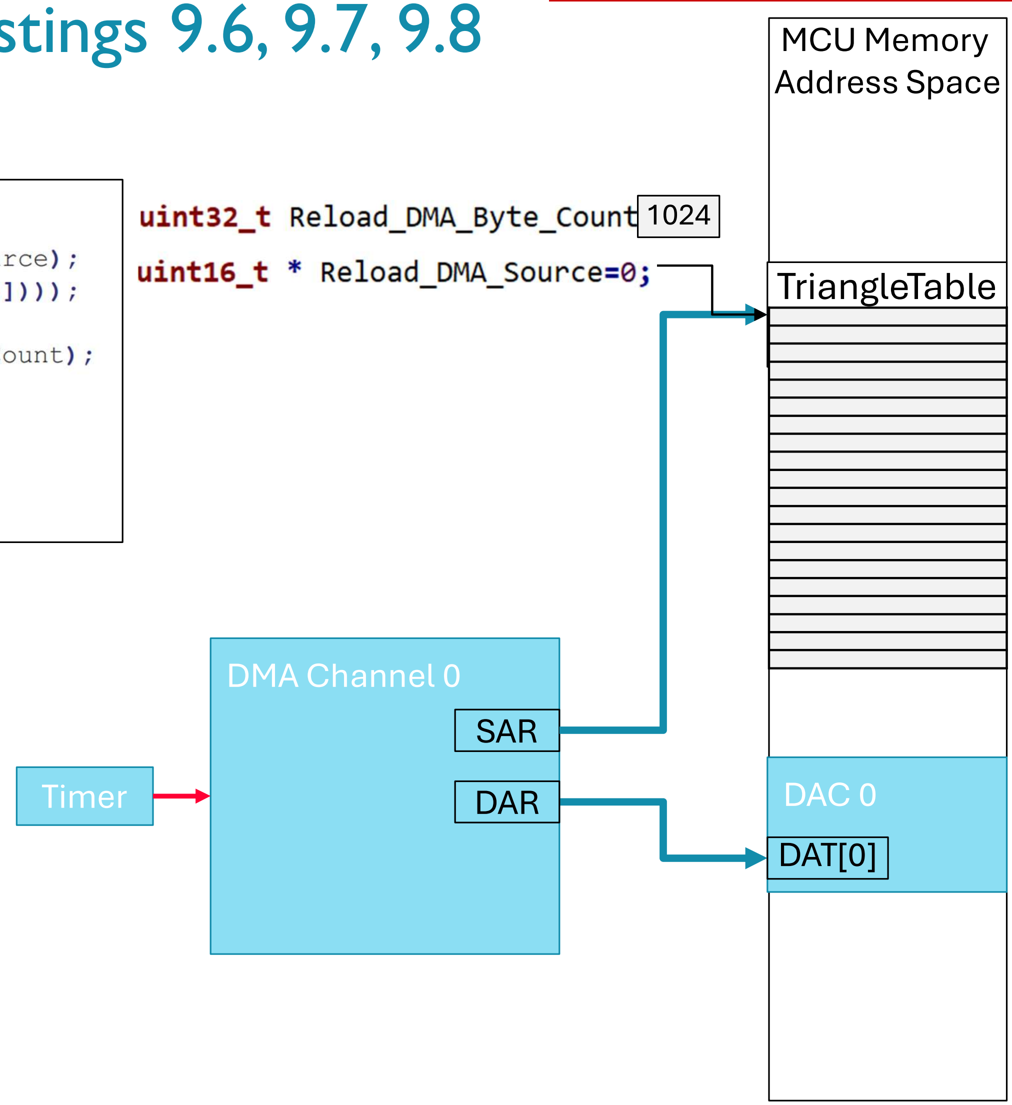
TriangleTable

```
void Init_DMA_For_Playback(uint16_t * source, uint32_t count) {
	// Save reload information
	Reload_DMA_Source = source;
	Reload_DMA_Byte_Count = count*2;

	// Gate clocks to DMA and DMAMUX
	SIM->SCGC7 |= SIM_SCGC7_DMA_MASK;
	SIM->SCGC6 |= SIM_SCGC6_DMAMUX_MASK;

	// Disable DMA channel to allow configuration
	DMAMUX0->CHCFG[0] = 0;

	// Generate DMA interrupt when done
	// Increment source, transfer words (16 bits)
	// Enable peripheral request
	DMA0->DMA[0].DCR = DMA_DCR_EINT_MASK | DMA_DCR_SINC_MASK |
	DMA_DCR_SSIZE(2) | DMA_DCR_DSIZE(2) | DMA_DCR_ERQ_MASK | DMA_DCR_CS_MASK;

	// Configure NVIC for DMA ISR
	NVIC_SetPriority(DMA0_IRQn, 2);
	NVIC_ClearPendingIRQ(DMA0_IRQn);
	NVIC_EnableIRQ(DMA0_IRQn);

	// Set DMA MUX channel to use TPM0 overflow as trigger
	DMAMUX0->CHCFG[0] = DMAMUX_CHCFG_SOURCE(54);
}
```

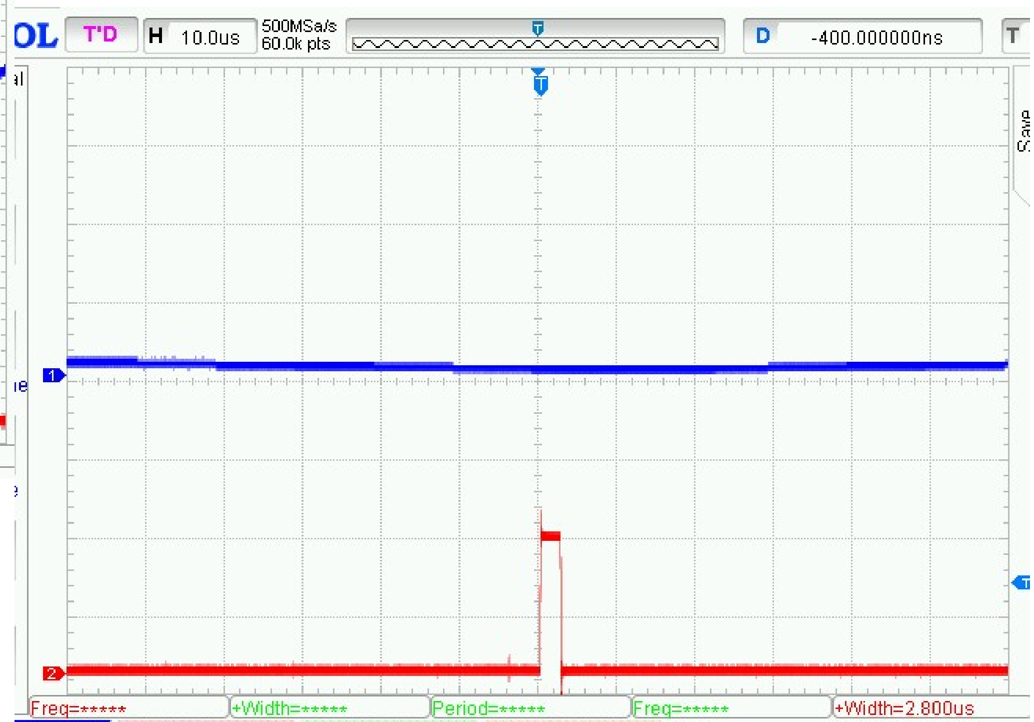
Example Source Code from ESF: Listings 9.6, 9.7, 9.8

```
void Start_DMA_Playback() {
      // initialize source and destination pointers
      DMA0->DMA[0].SAR = DMA_SAR_SAR((uint32_t) Reload_DMA_Source);
      DMA0->DMA[0].DAR = DMA_DAR_DAR((uint32_t) (&(DAC0->DAT[0])));
      // byte count
      DMA0->DMA[0].DSR_BCR = DMA_DSR_BCR_BCR(Reload_DMA_Byte_Count);
      // clear done flag
      DMA0->DMA[0].DSR_BCR &= ~DMA_DSR_BCR_DONE_MASK;
      // set enable flag
      DMAMUX0->CHCFG[0] |= DMAMUX_CHCFG_ENBL_MASK;
}
```

```
void DMA0_IRQHandler(void) {
   // Turn off blue LED in DMA IRQ handler
   Control_RGB_LEDs(0,0,0);
   // Clear done flag
   DMA0->DMA[0].DSR_BCR |= DMA_DSR_BCR_DONE_MASK;
   // Start the next DMA playback cycle
   Start_DMA_Playback();
   // Turn on blue LED
   Control_RGB_LEDs(0,0,1);
}
```



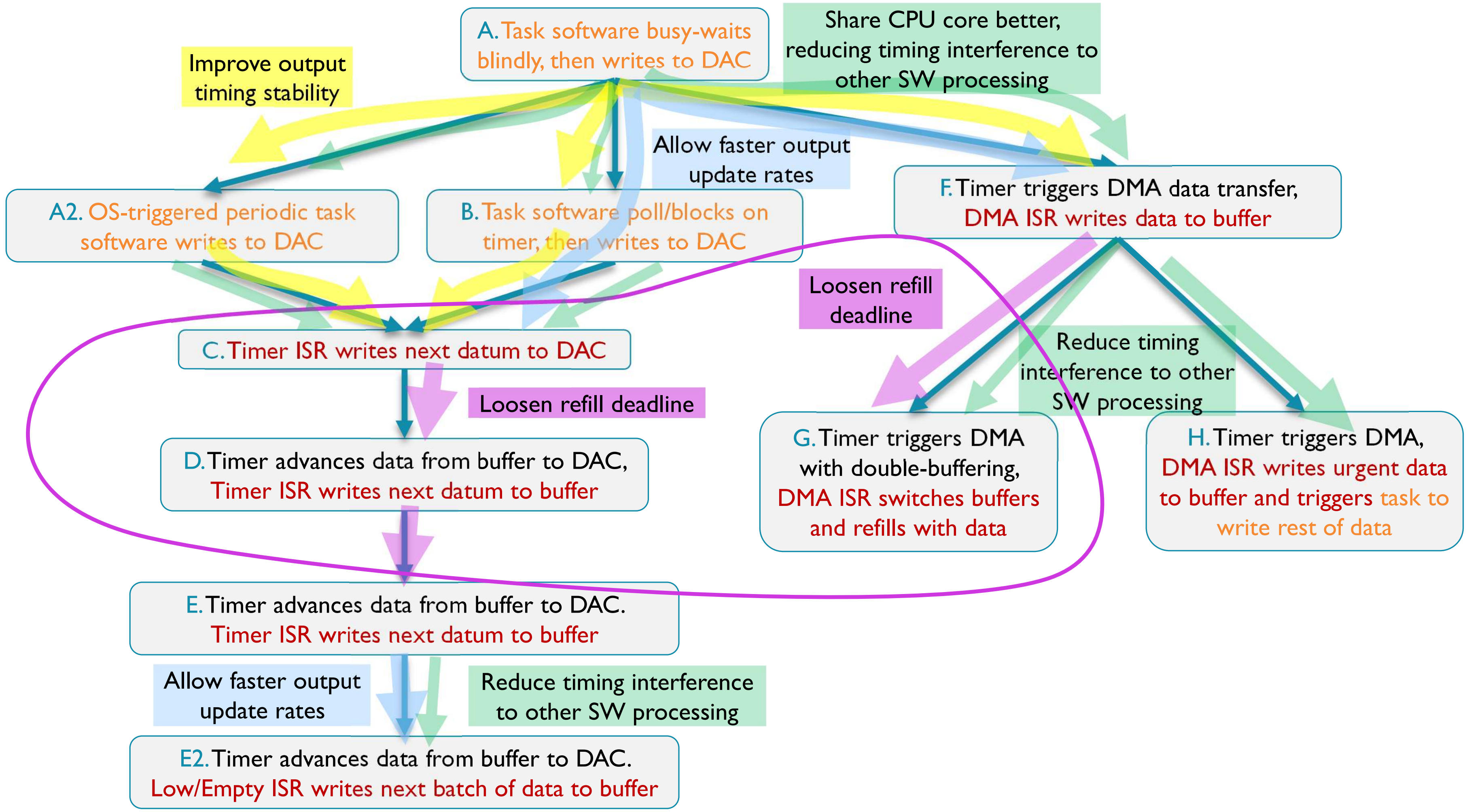
F. Timing Analysis



LOOSENING DEADLINES

Share CPU core better, reducing timing interference to other SW processing

Overview of Waveform Generator Design Evolution: What and Why



Detailed Overview of What and Why

A. Task software writes to DAC

*Output timing bad: Very unstable, vulnerable to other software (processes and handlers), timing errors accumulate. **Greedy**, doesn't share CPU.*

Add HW timer (tracks time accurately)

And add OS with ticks from HW timer interrupts

And access HW timer directly

And add DMA with ISR, software buffer

A2. OS-triggered periodic task software writes to DAC

B. Task software poll/blocks on timer, then writes to DAC

F. Timer triggers DMA data transfer, DMA ISR writes data to buffer

Output timing better: Tolerates more interference, vulnerable to processes and handlers, errors don't accumulate. Shares CPU. Tight deadline: T_{Sample}

Put code in HW timer's ISR

Output timing better: Tolerates more interference, vulnerable to processes and handlers, errors don't accumulate. Greedy, doesn't share CPU. Tight deadline: T_{Sample}

1. Tight Deadline: ISR must write first new sample to buffer within T_{Sample}

2. Long DMA ISR is delays other processing too much

C. Timer ISR writes next data to DAC

Output timing: Even better. Vulnerable to other ISRs and interrupt locking f_{sample} times per second Tight deadline: T_{Sample}

Use 1-deep DAC input buffer

Split into double-buffer to ease first sample's deadline and cuts ISR duration in half.

Move non-urgent work to task

D. Timer advances data from buffer to DAC, Timer ISR writes next data to buffer

G. Timer triggers DMA with double-buffering, DMA ISR switches buffers and refills with data

H. Timer triggers DMA, DMA ISR writes urgent data to buffer and triggers task to write rest of data

Deadline better: $2T_{Sample}$ Interrupt overhead for each sample wastes CPU time.

Use N-deep DAC input buffer with low/empty ISR

E. Timer advances data from buffer to DAC. Timer ISR writes next data to buffer

Interrupt overhead for each sample wastes CPU time

Add N-deep DAC input buffer with low/empty ISR

E2. Timer advances data from buffer to DAC. Low/Empty ISR writes next batch of data to buffer

E. Add Multi-Entry Buffer for DAC

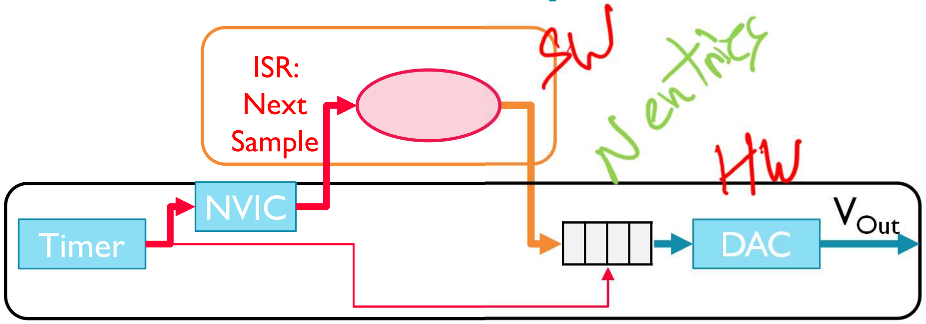
Similar timing

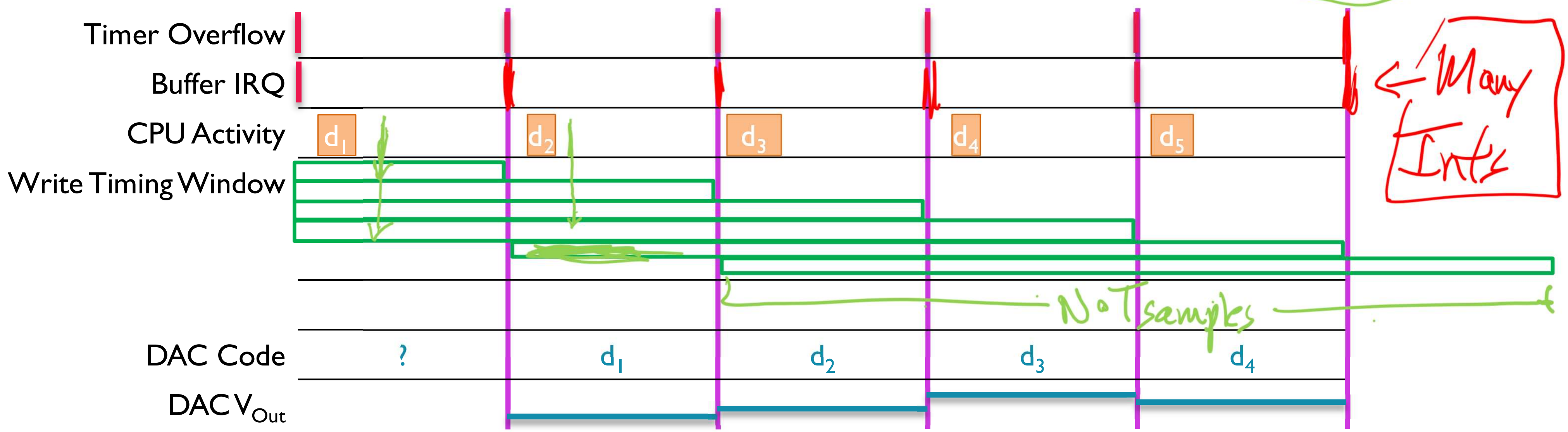
```
(code) {
}
```

Diff. Timing

often & fast

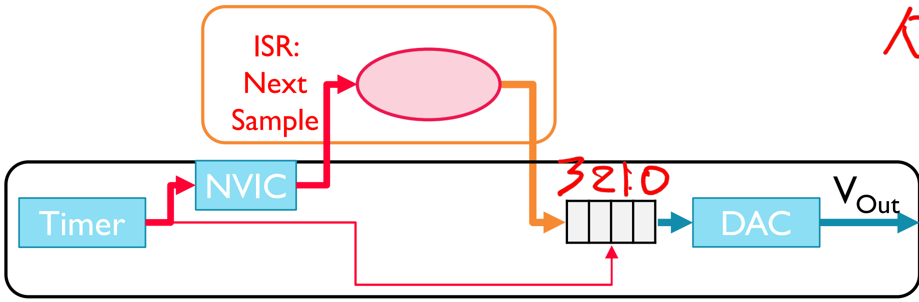
slow, rarely exec

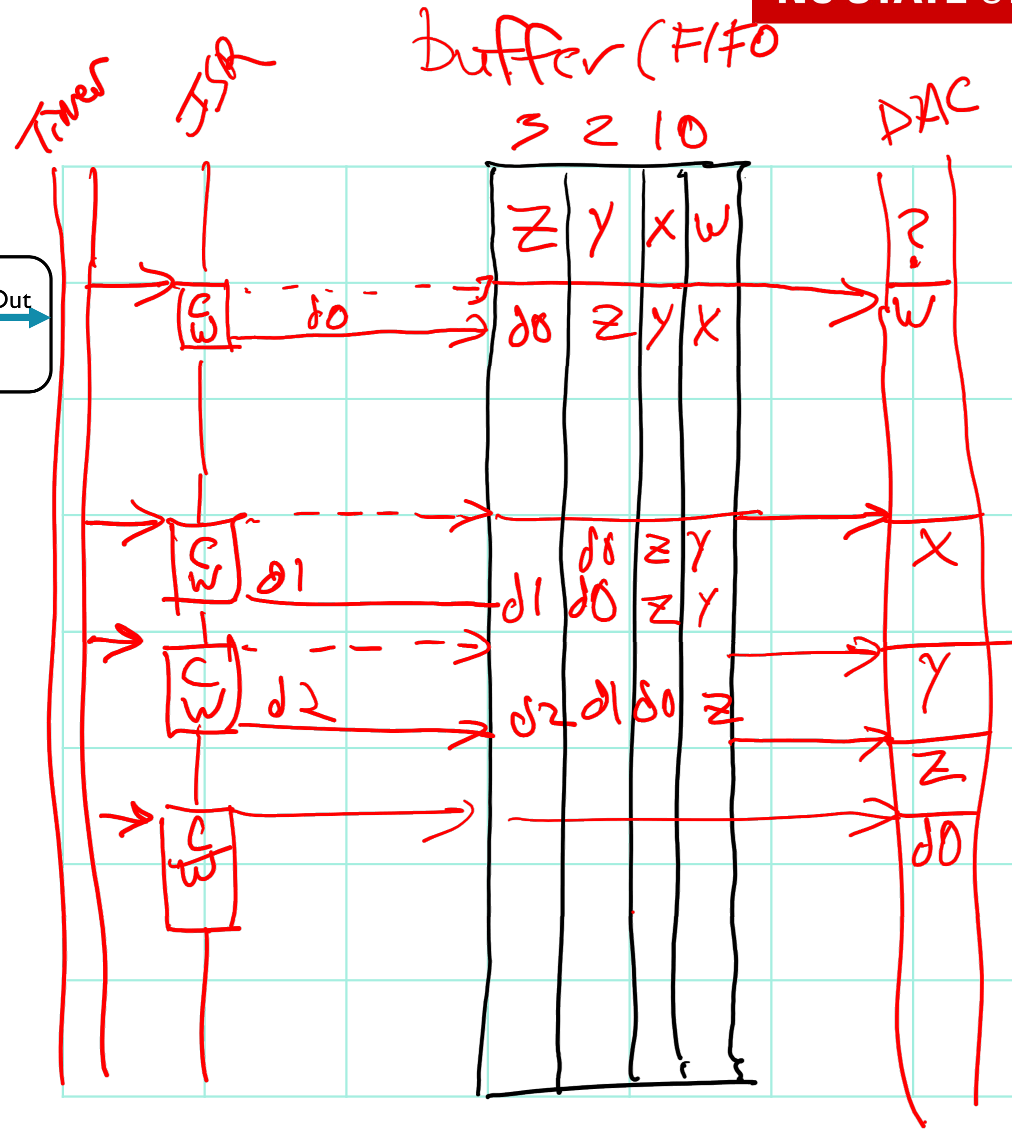




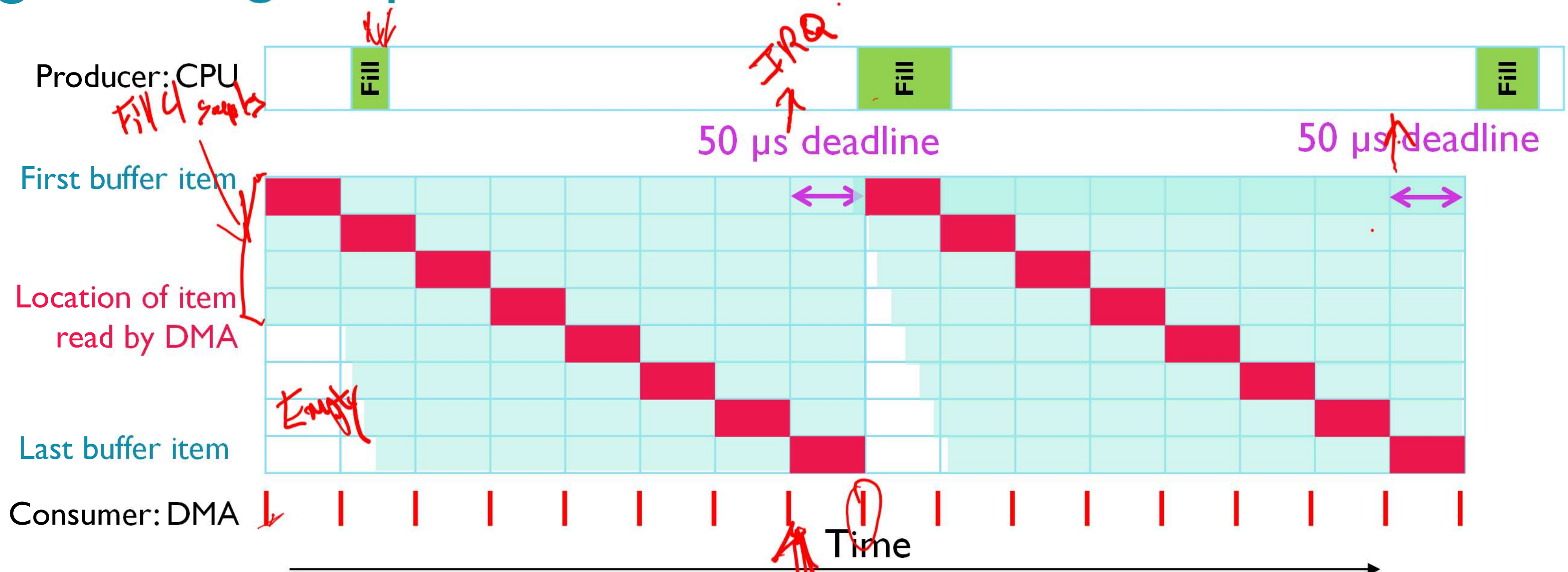
- Buffer is queue: first-in, first-out (FIFO)
- Buffering delays data, so will not work for all applications

Sequence Diagram



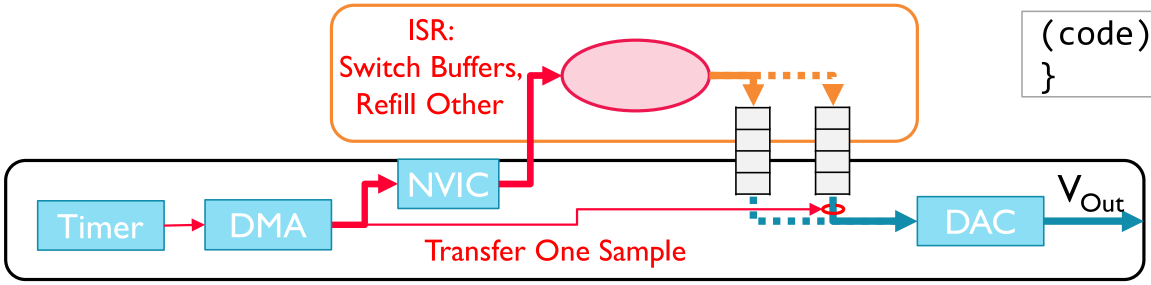


Tight Timing Requirements for Buffer



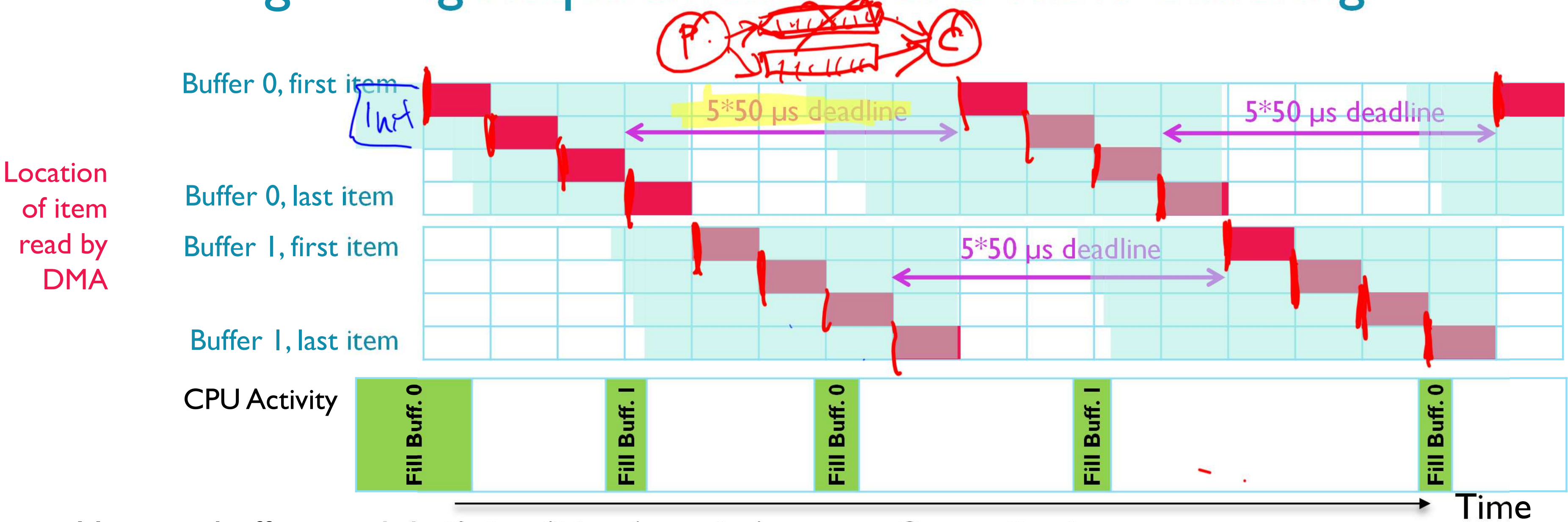
- Despite fewer interrupts (due to multiple buffer entries), still need to save next sample to buffer **before** DMA reads it
- Examine and understand timing requirements for buffer
- Producer adds data (light blue-green) to fill in buffer
  - In example, first four items have already been added
- Consumer reads data item from red buffer entry
  - Data item in buffer is not needed (old, stale) after being read by consumer
- Producer (Thread_Refill_Sound_Buffer) must stay ahead of consumer (DMA controller)

G. DMA with Double Buffering




```
(code) {
}
```

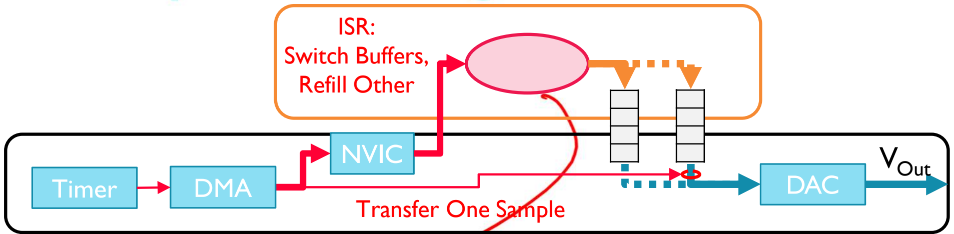
Loosening Timing Requirements with Double-Buffering



- Use two buffers, each half-size (N = 4 entries)
- Initialization
  - Start filling buffer 0
  - Can start playing buffer 0 after it has ≥1 sample
- After buffer 0 is filled, start filling buffer 1,
- Operation: After playing last sample from buffer 0,
  - Switch to playing buffer 1
  - Start refilling buffer 0
- Generalization
  - After playing last sample from buffer x, switch to playing buffer y, start filling buffer x
- Deadlines
  - Now have two deadlines, one per buffer ☹
  - Much looser deadlines: extended to from T_{Sample} to $(N+1)*T_{Sample}$ ☺

NC STATE UNIVERSITY

Sequence Diagram



- Switch me to other buffer
- Refill this One