Lecture 06
Timers

CE346 – Microprocessor System Design

Branden Ghena – Spring 2021

Some slides borrowed from:
Josiah Hester (Northwestern), Prabal Dutta (UC Berkeley)

Northwestern

Today’s Goals

- Understand the role of clocks in a microcontroller

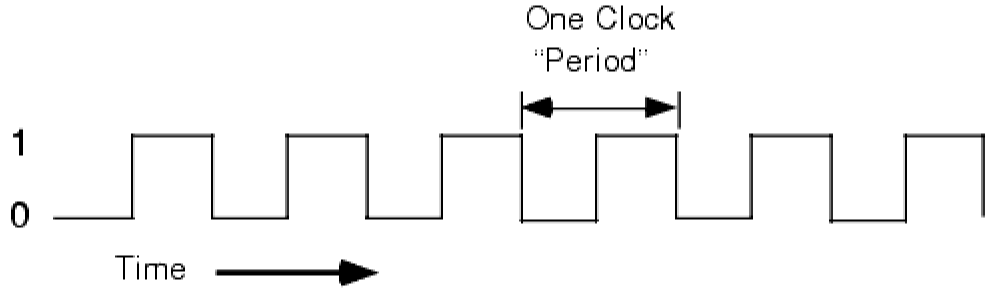
- Explore functionality of various timer peripherals on the Microbit

Outline

- **Clocks**
- Timers
- Real-Time Counter
- Watchdog

What are clocks?

- Clock signals, in the microcontroller context, are oscillating square wave signals used to latch inputs

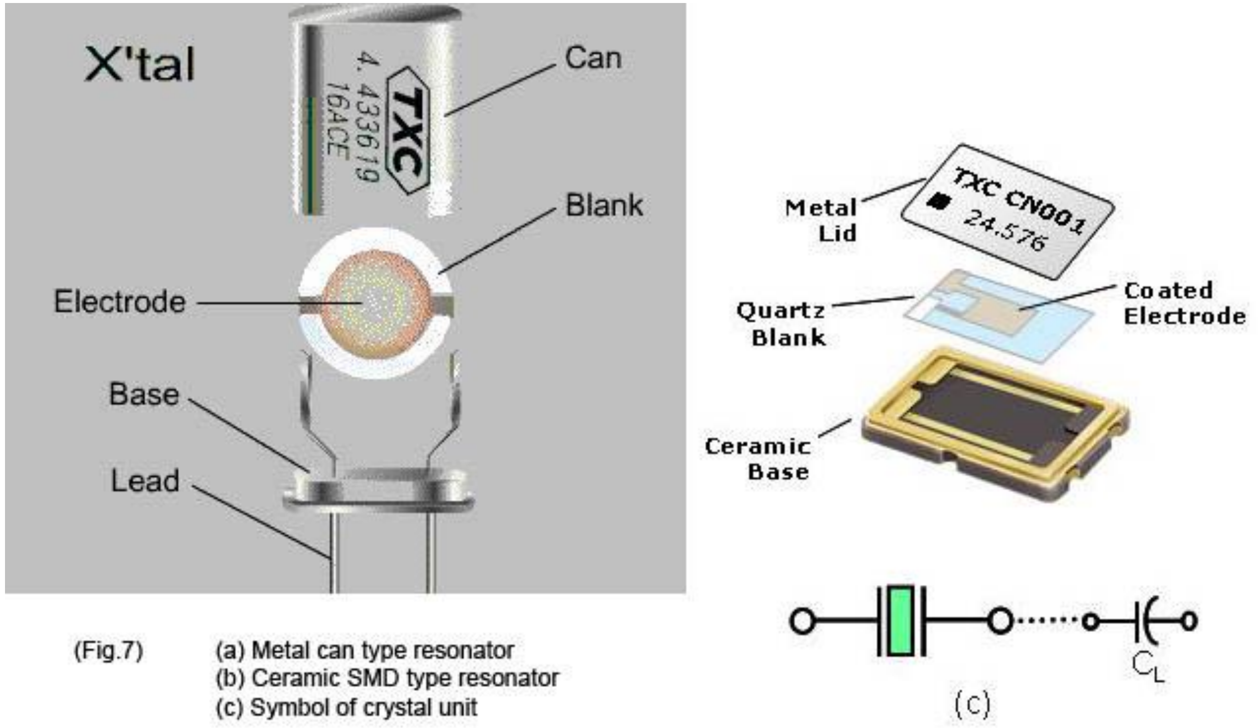


- A clock MUST be running for (almost) anything on a microcontroller to function (processor and peripherals)
  - Exceptions:
    - Low-power input interrupts
      - GPIOTE port interrupt, Analog LPCOMP interrupt, NFC sense interrupt, USB power interrupt
    - Reset signal

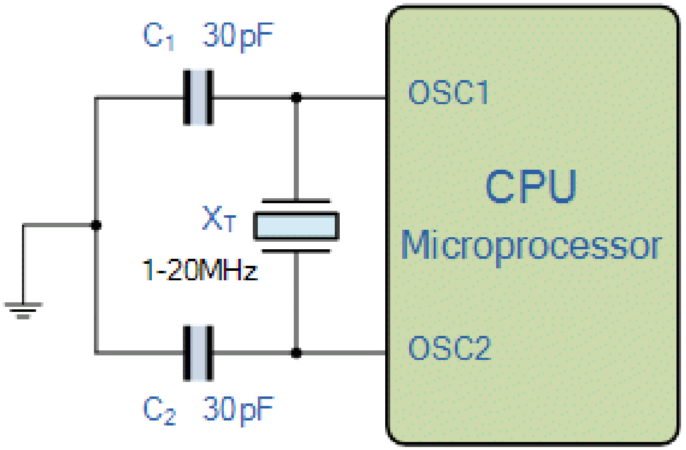
Generating clocks

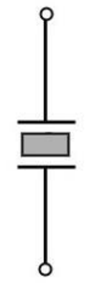
- External crystal oscillator
  - Creates clock signal
  - Chunk of quartz
  - Behaves like RLC circuit but uses less energy

- Internal mechanisms
  - RC oscillator
    - Creates clock signal
    - Less accurate and higher energy than crystal
  - Phase-Locked Loop (PLL)
    - Multiply input to create new higher frequency clocks

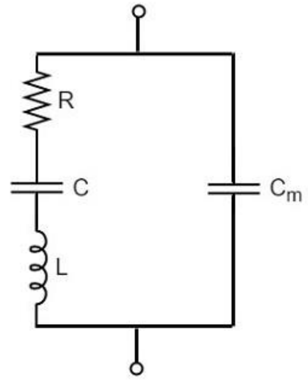


(Fig.7) (a) Metal can type resonator
(b) Ceramic SMD type resonator
(c) Symbol of crystal unit





Symbol of a Crystal

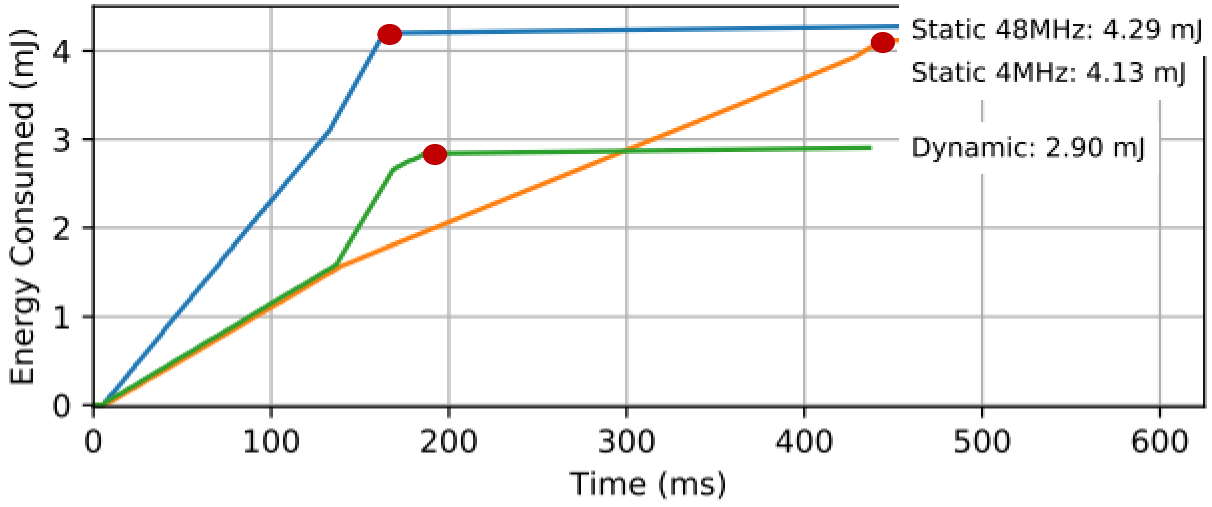


Equivalent circuit of a crystal

Microbit crystal for nRF52833

Clocks and energy

- Fundamental tradeoff
  - Faster clock gets things done faster but uses more energy
  - Slower clock uses less energy but gets things done slower
  - Which to use depends on the situation
    - CPU bound: faster clock, IO bound: slower clock



Example of clock selection for a mixed load (part IO, part CPU)

Energy consumed becomes a horizontal line when the task is completed

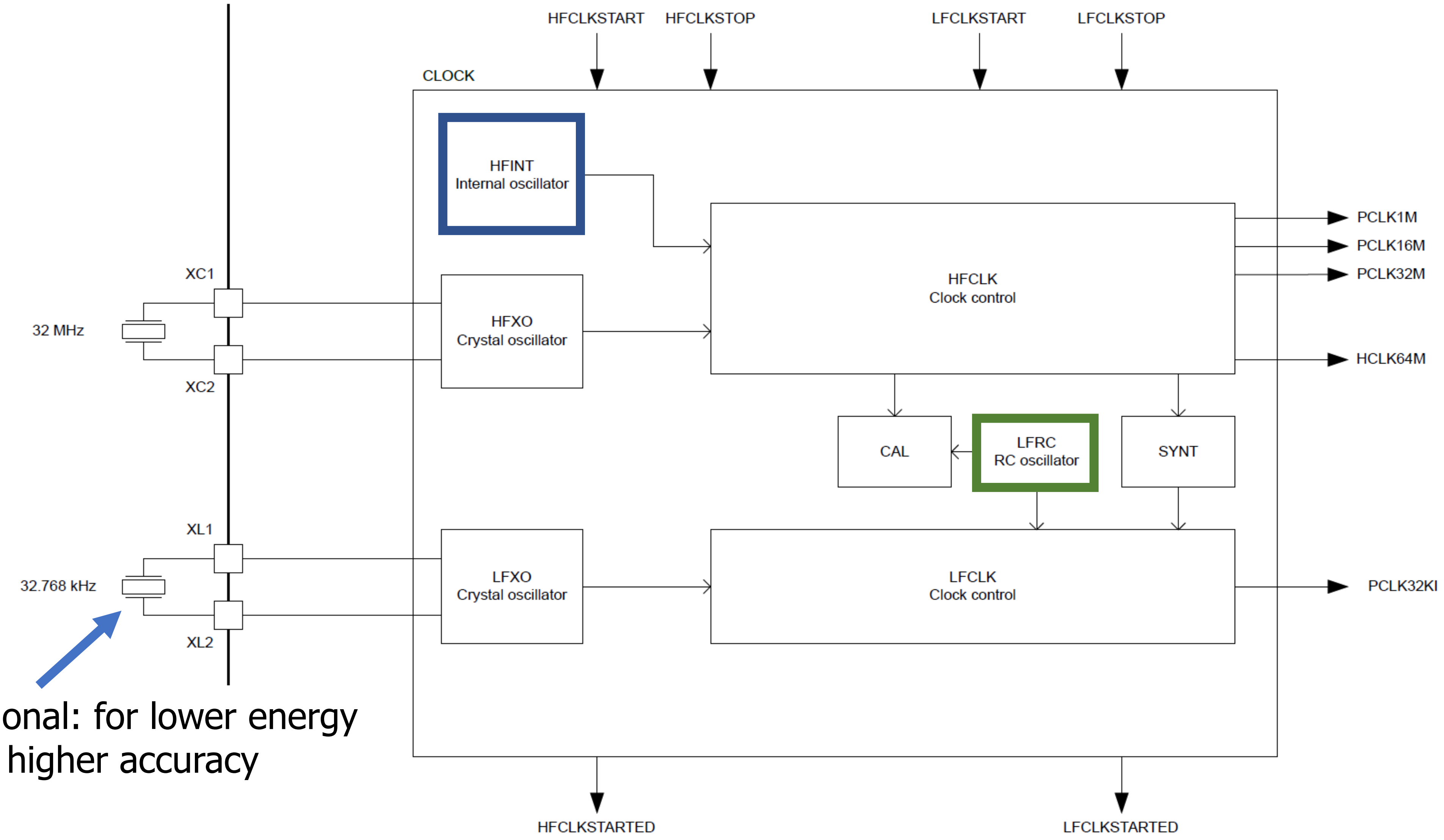
Chiang et al. "Power Clocks: Dynamic Multi-Clock Management for Embedded Systems" EWSN 2021

Controlling clocks

- Some microcontrollers provide extremely fine-grained control over clocks
  - Really complicated section of code to get working
  - Many combinations are invalid
  - Manually enable/disable clocks as needed

- nRF52 instead gives almost no control but is easier to use
  - One 64-MHz clock for processor
  - Multiple peripheral clocks, but (most) peripherals are hardwired to one
    - 16 MHz for almost all peripherals (PDM and I2S are 32 MHz)
  - Low-frequency 32 kHz clock for low-power peripherals
  - Automatically enables/disables clocks

nRF52833 clocks



Optional: for lower energy and higher accuracy

Electrical characteristics

- Active power of clocks
  - 32 kHz crystal run current: 0.23 µA
  - 32 kHz RC oscillator run current: 0.70 µA
  - 32 MHz crystal average run current: 300-700.00 µA
  - 32 MHz standby current: 110.00 µA

- Startup time for external crystals
  - 32 kHz crystal: 250-500 ms (milliseconds!!!)
  - 32 MHz crystal: 60-200 µs
  - Beware: switching can lead to delays and instability
    - nRF52 uses RC oscillator while crystal is not yet ready

Outline

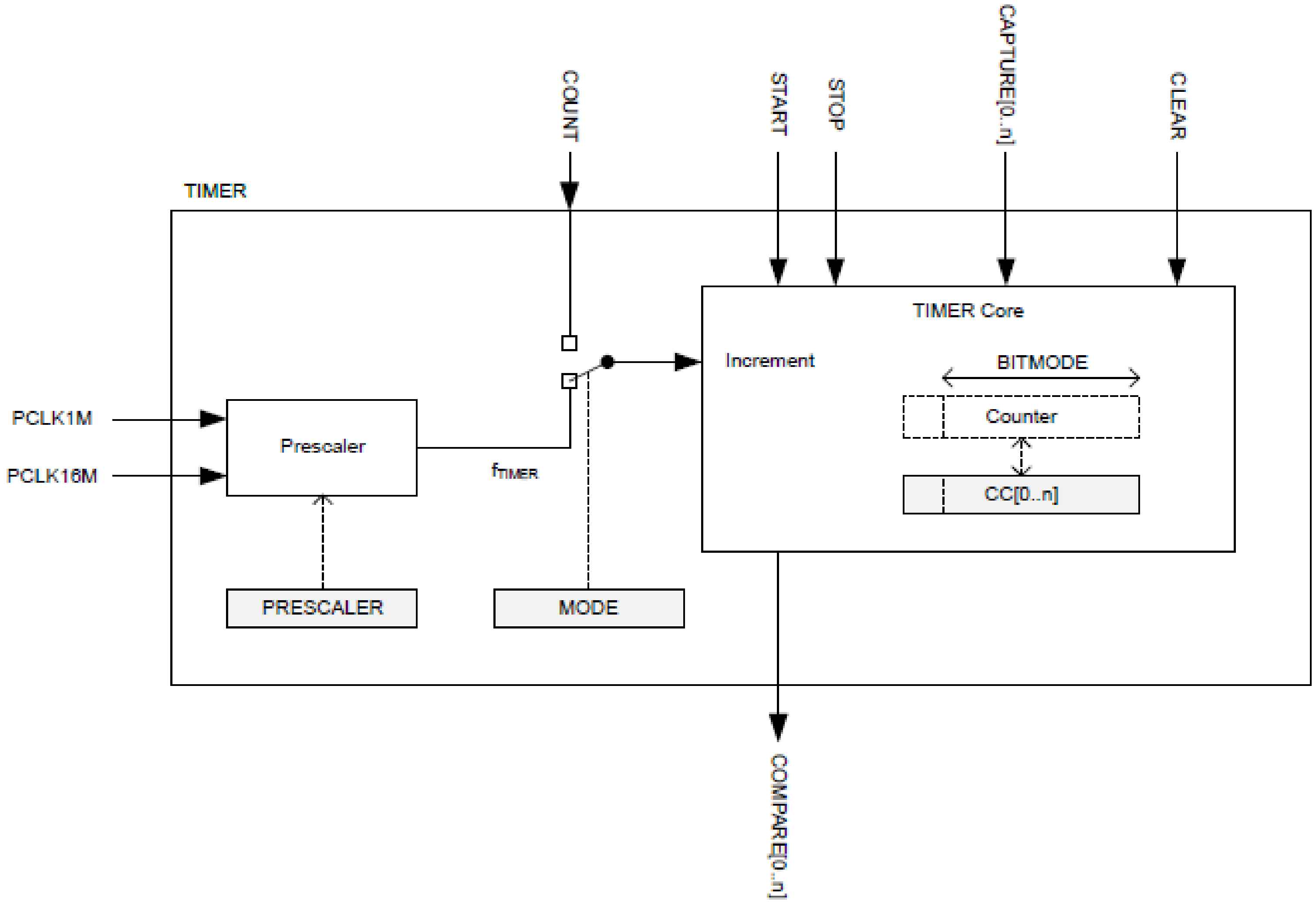
- Clocks
- **Timers**
- Real-Time Counter
- Watchdog

Timer peripherals

- Common need for embedded systems: sense of time
  - Start this behavior after a certain amount of time
  - Stop this behavior after a certain amount of time
  - Measure how much time passed between two events

- Timer peripherals
  - Input is one of the system clocks
  - Counts up a register at each clock tick
    - Looking at register at start and end can give real-world duration
  - Compare to saved value and trigger interrupt on match
    - Allows interrupts to be scheduled in the future

Timer peripheral on nRF52833



Input and Prescaler

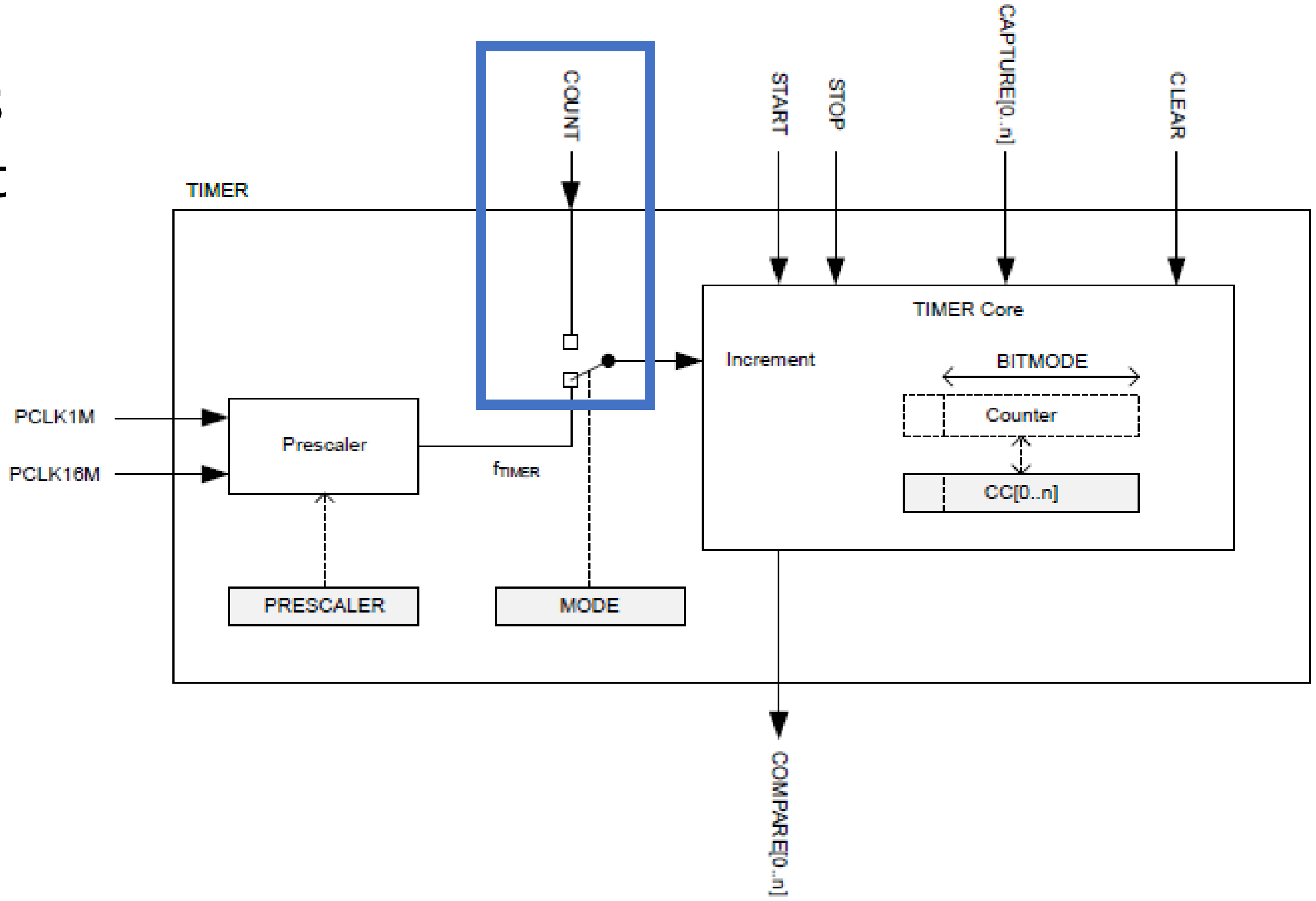
$$f_{\text{TIMER}} = \frac{16\text{ MHz}}{2^{PRESCALER}}$$

- Prescaler is a 4-bit number
  - Possible timer input clocks: 16 MHz – 488 Hz
- Ticks counted with (up to) 32-bit internal Counter:
  - Minimum 268 seconds until overflow (at 62.5 ns per tick)
  - Maximum 101 days until overflow (at 2.04 ms per tick)

Alternate input source for counter mode

- Counter mode works with non-timer inputs
  - E.g. GPIO input event

- Count anything!



Capture/Compare registers (CC)

- 32-bit storage registers (each timer has multiple)
    - Uses: capturing or comparing

- On Capture[n] event
    - Internal Counter value copied to CC[n]

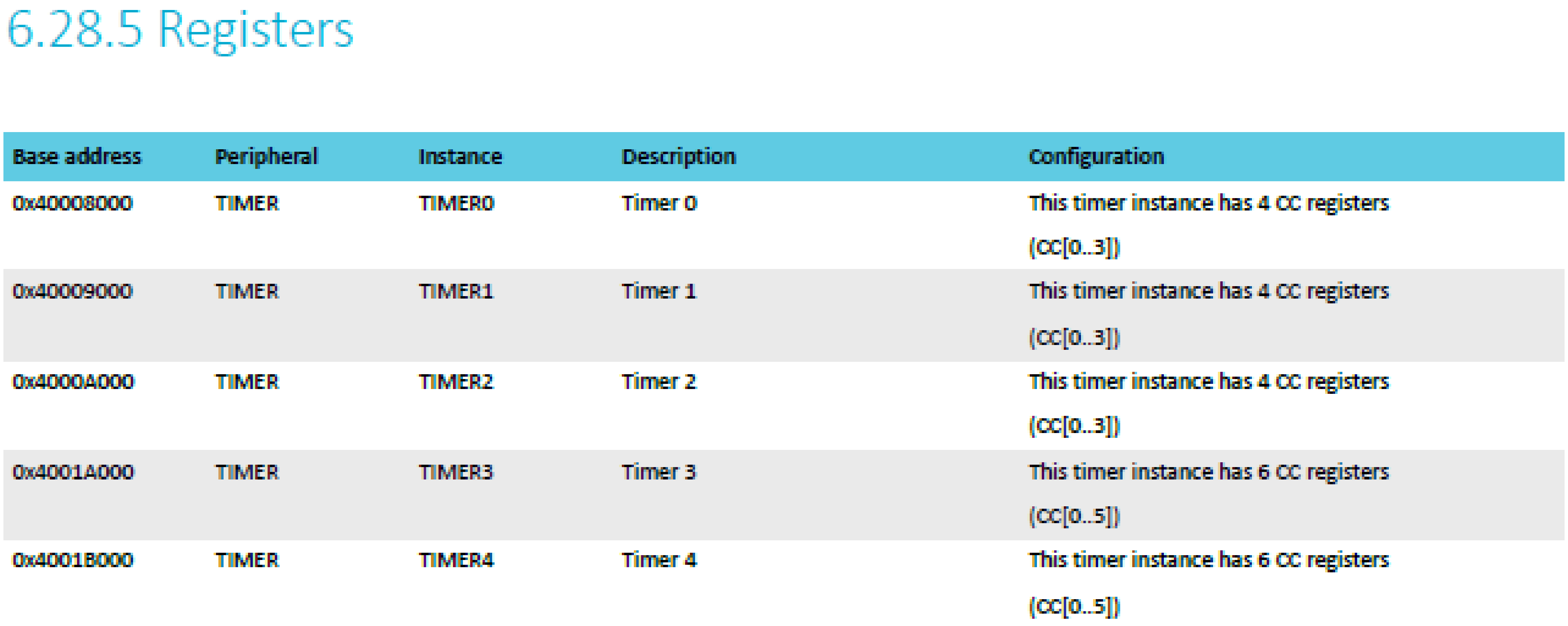
- Capture used to measure durations of events
    - Capture can be triggered by software or by Events from other peripherals
    - Multiple registers to measure multi-part events

Comparing with CC registers

- When internal Counter value equals a CC register
  - Corresponding Compare[n] event is triggered
  - Can trigger interrupts

- Usually written to in advance to start/stop behavior
  - Toggle LED every second
  - Sample sensor every five minutes
  - Refresh LED matrix every 1/60 seconds

The nRF52833 has multiple Timer instances



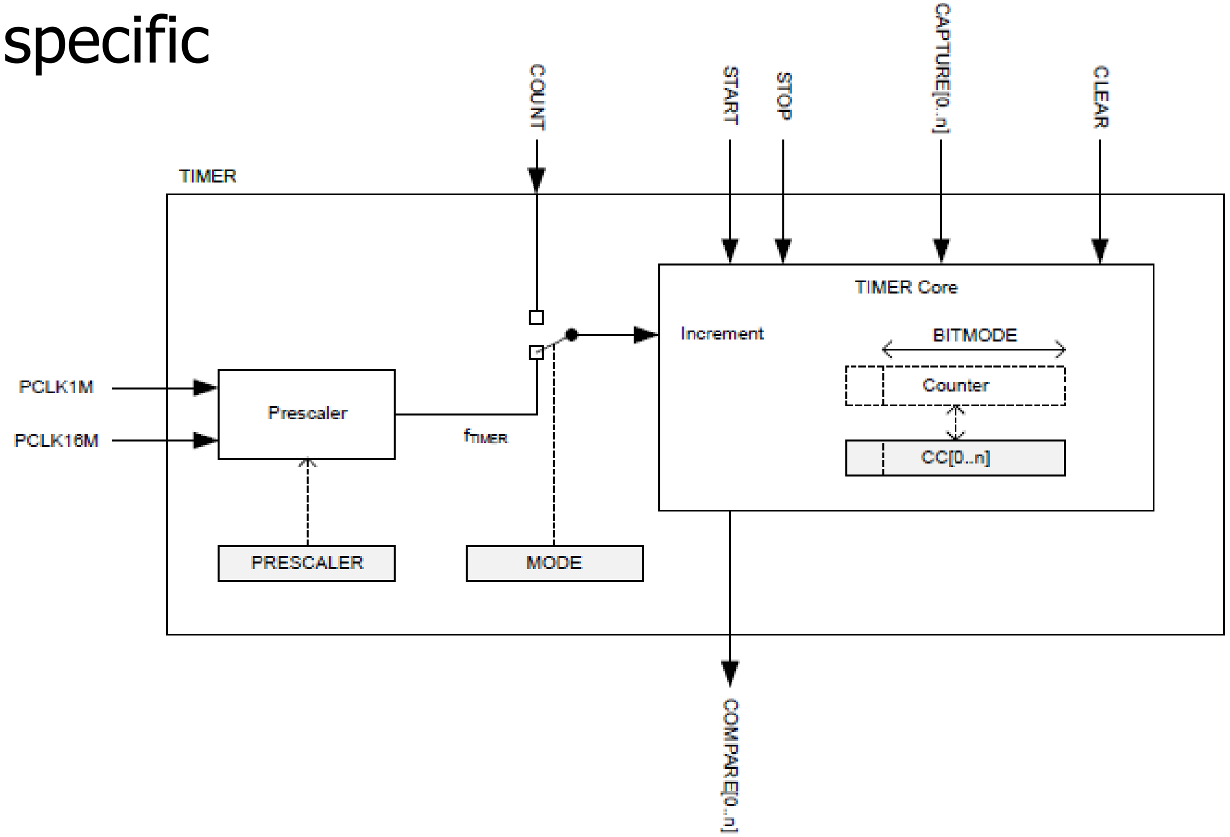
6.28.5 Registers

Base address	Peripheral	Instance	Description	Configuration
0x40008000	TIMER	TIMER0	Timer 0	This timer instance has 4 CC registers (CC[0..3])
0x40009000	TIMER	TIMER1	Timer 1	This timer instance has 4 CC registers (CC[0..3])
0x4000A000	TIMER	TIMER2	Timer 2	This timer instance has 4 CC registers (CC[0..3])
0x4001A000	TIMER	TIMER3	Timer 3	This timer instance has 6 CC registers (CC[0..5])
0x4001B000	TIMER	TIMER4	Timer 4	This timer instance has 6 CC registers (CC[0..5])

Bonus concept: shorts

- Reminder: Tasks are inputs and Events are outputs
- Shorts connect an Event to a Task within a peripheral
  - Tasks and Events are fairly nRF specific

- Timer shorts
  - Connect Compare[n] to Clear
  - Connect Compare[n] to Stop



Usage: how do we set a one second timer?

- Assume timer is already running

1. Get current time from timer
2. Add 1 second worth of ticks to it
    - $\frac{16000000}{2^{PRESCALER}}$ is the number of ticks per second
3. Set an unused Compare register to value
4. Enable interrupts for that Compare event

Warning: what if you’re setting a 1 us timer instead? Or a 100 ns timer?

Timer could expire *before* software writes it to the peripheral.

Check your understanding

- Prescaler value is 4
- Current internal Counter value is 0x1000
- Want a 0.5 second timer

$$f_{\text{TIMER}} = \frac{16\text{ MHz}}{2^{PRESCALER}}$$

- **What do you set the CC[0] register to?** (32-bits)

Check your understanding

$$f_{\text{TIMER}} = \frac{16\ \text{MHz}}{2^{PRESCALER}}$$

- Prescaler value is 4
- Current internal Counter value is 0x1000
- Want a 0.5 second timer

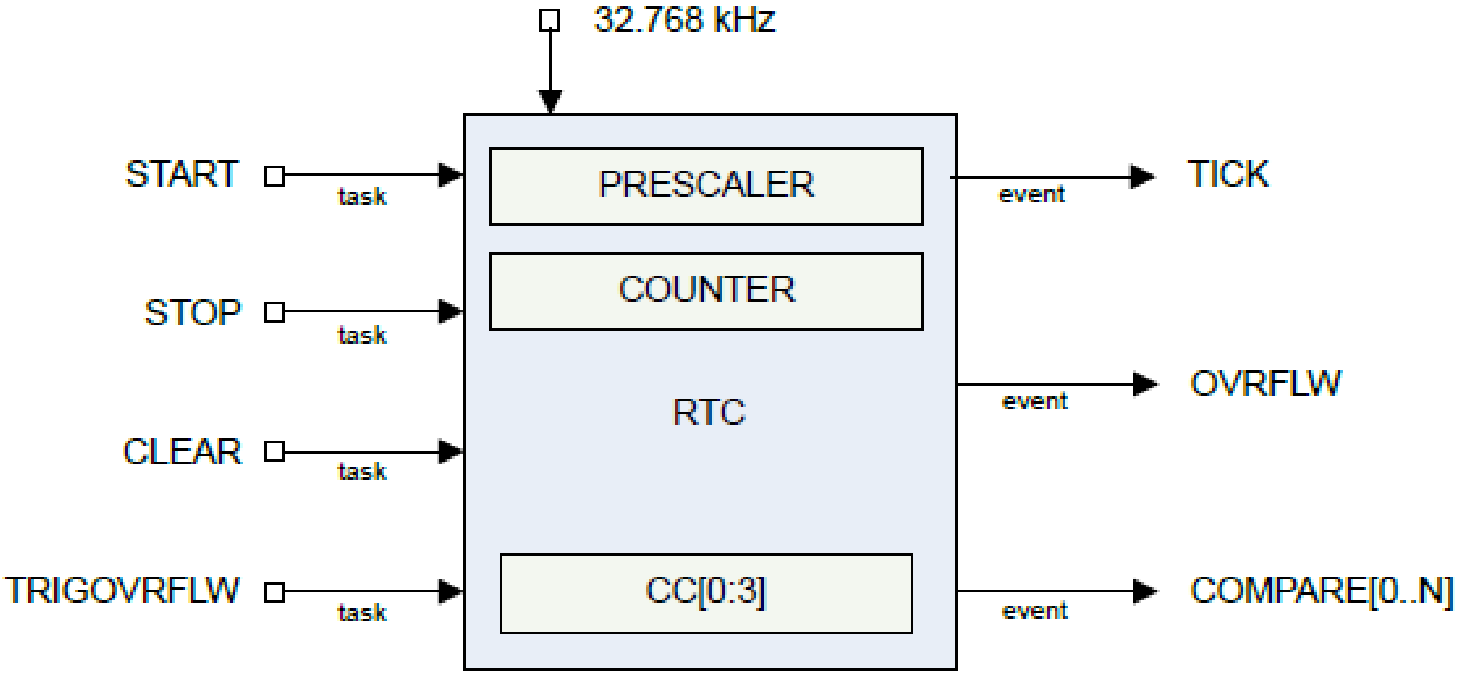
- **What do you set the CC[0] register to?** (32-bits)
  - 1 MHz Timer frequency -> 500,000 ticks in 0.5 seconds
  - 500000 -> 0x7A120
  - Plus initial value of counter = **0x7B120**

Outline

- Clocks
- Timers
- **Real-Time Counter**
- Watchdog

Real-time Counter

- Low-power (32 kHz) version of Timer
  - Only a 24-bit internal Counter



- Note: abbreviated RTC, but that already means something else (Real-Time Clock)

Differences between Real-Time Counter and Timer

- Runs off of LFCLK instead of HFCLK
  - With smaller prescaler value (4096 vs 32768)

- 24-bit counter vs 32-bit counter for Timer

- Can read the Counter value directly
  - No need for Capture task

- Otherwise extremely similar. Just a low-power version of Timer

Time resolution for Real-Time Counter

$$f_{\text{TIMER}} = \frac{32\ \text{KHz}}{Prescaler+1}$$

- Resolution
    - Minimum: 30.517 µs, overflows in 512 seconds (24-bit Counter)
    - Maximum: 125 ms, overflows in 582 hours

- Not as precise as the Timer (62.5 ns best precision)
    - Possible design: use both
        - Real-Time Counter for most of the waiting
        - Chained into Timer for precise remaining amount of time

Outline

- Clocks
- Timers
- Real-Time Counter
- **Watchdog**

Reliable systems

- What’s the most common way to solve computer problems?
    - Turn it off and turn it on again.

    - **Why?**

Reliable systems

- What’s the most common way to solve computer problems?
  - Turn it off and turn it on again.

  - **Why?**

  - Resets “state” to original values, which are likely good
    - Startup is often well-tested

    - It’s long-running code interacting in unexpected ways that leaves systems in a broken state

Watchdog timer (WDT)

- Focused on failures where the system “hangs” forever
  - Maybe software, maybe hardware!
- Can’t know for certain the system is hung, but can know practically
  - Select a timeout that is the maximum amount of time you expect the system to ever go without looping in main()
  - Multiply it by 2-10
  - Set a watchdog timer to that value
- If watchdog timer ever expires, it resets the system (in hardware)

Watchdog configuration

$$\text{timeout (seconds)} = \frac{Counter\ Reload\ Value + 1}{32768}$$

- Configure watchdog
    - Can choose whether to count down during Sleep mode or Debug mode

- Set a Counter Reload Value (CRV, 32-bits)

- Start the watchdog timer
    - Loads internal Counter to CRV value
    - Starts counting down at 32 kHz

Running applications with a watchdog timer

- Need to periodically reset the watchdog to keep it from expiring
  - Known as “feeding” the watchdog or “kicking” the watchdog

- Reload Request register
  - Must write sequence 0x6E524635 to reload watchdog
  - Incredibly unlikely to happen by accident

- While running, watchdog is protected from modification
  - Configure once, run forever (at least until a reboot)
  - Only option is to make periodic Reload Requests

- Default off on the nRF52833 (default on for the MSP430!)

Outline

- Clocks
- Timers
- Real-Time Counter
- Watchdog