Run-Time Methods for Making Embedded Systems Robust

Lecture 21

Today

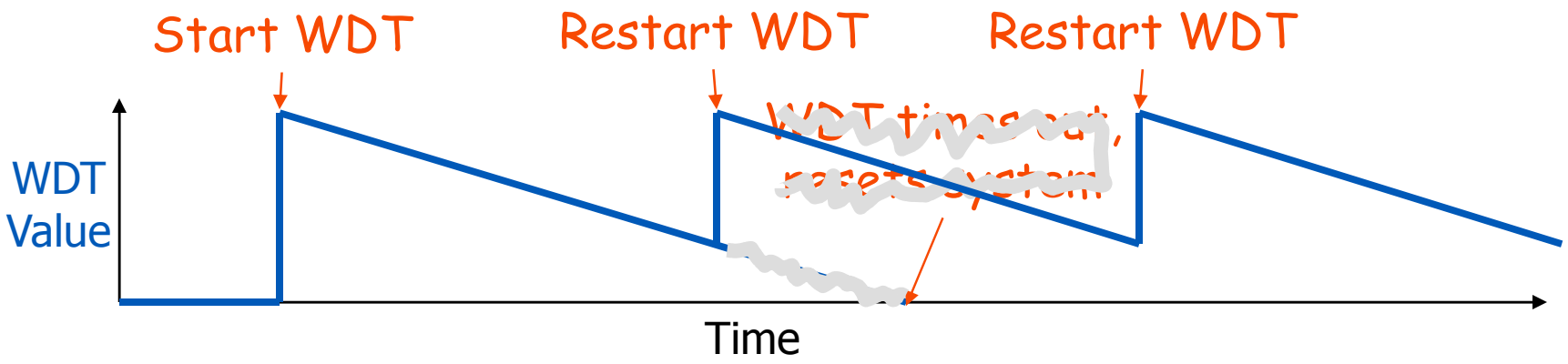
Need to make embedded systems robust

- Implementation flaws: Code may have implementation bugs
- Design flaws: Real world may not behave the way we expected and designed for
- Component failures: Sometimes things break

Run-time mechanisms for robust embedded systems

- Watchdog timer
- Stack-pointer monitor
- Brown-out detector

Watchdog Timer Concepts (WDT)



Goal: detect if software is not operating correctly

Assumption: healthy threads/tasks will periodically send a heartbeat ("I'm alive") signal

Mechanism

- Use heartbeat signals from tasks to restart a timer
- If timer ever expires, the system is sick, so reset

Typically used as a final, crude catastrophic mechanism for forcing system software back into known state

Time-Out Actions

Simple solution: reset entire system

- May need to explicity toggle reset pin to ensure CPU is fully reset (rather than just jumping to reset ISR)
- Reset should configure all I/O to safe state

NMI Solution: generate non-maskable interrupt for debug

- Use NMI ISR to save picture of CPU and thread state
- Can then examine what happened with debugger or in-circuit emulator

WDT Time-Out flag in memory

- Set flag upon time-out before reset
- Examine this bit in reset ISR to determine whether to boot system normally or with debug mode (without overwriting RAM)

Resetting the WDT in a Multithreaded Application

Each periodic task updates a timestamp when it starts running

WDT thread checks timestamp for each thread *i* to make sure it was run no more than T_i ago. If all threads are ok, restart the WDT.

Any problems with this?

Why not put it into the scheduler?

Design Suggestions for WDT

- WDT should be driven by a reliable clock (if clock fails, or is incorrectly initialized, WDT will never expire)
- WDT should be difficult to accidentally reset
  - Limit reset to accessing one address, rather than range
  - Require sequence of commands
- WDT should be difficult to accidentally disable
- Should be able to disable WDT externally with a very obvious jumper (use to simplify debugging)
- Choose WDT period appropriately
  - Too long and system is out of control long enough to get into real trouble
  - Too short and you need to reset WDT frequently in your code (code writing and analysis overhead)

M30262 Watchdog Timer (WDT)

15-bit down counter

- Decremented by prescaled BCLK clock signal
  - BCLK = clock which drives CPU. Is external clock divided by 2,4, 8 or 16
  - Prescaler divides BCLK by 16 or 128
  - Our board:
    - Prescale by 16: 20 MHz/(16*32768) = 38.1 Hz, 26.2 ms
    - Prescale by 128: 20 MHz/(128*32768) = 4.76 Hz, 210 ms
- Is preset to 7FFF by
  - Code writing to WDTS (000E)
  - RESET signal being asserted
  - WDT itself expiring
- Doesn't start counting until a write to WDTS (000E)

Counter reaches 0?

- Results in Oscillation Stop Detection/Watchdog Timer interrupt (non-maskable interrupt), vector is at FFFF0

A Code Example

```
void WD_Init(){                    //Initialize Watchdog Timer
   cm06 = 1;    //BCLK = (20/8) MHz = 2.5 MHz (Xin div by 8, default)
   wdc7 = 1;    //prescaler is div by 128,  Watchdog Timer
                //period = (32,768 x 128) / (2.5 MHz) = 1.678s
   wdts = 0;    //start Watchdog Timer by writing any value to wdts
                //reg (value always resets to 0x7fff when written to)
}

void TimerA1_ISR(void){
   wdts = 0;    // retstart watchdog timer
   red_led_status != red_led_status;
   if (red_led_status) red_led = 1;
        else red_led = 0;
   }
```

Mechanisms for robust embedded systems

Watchdog timer

Stack-pointer monitor

Brown-out detector

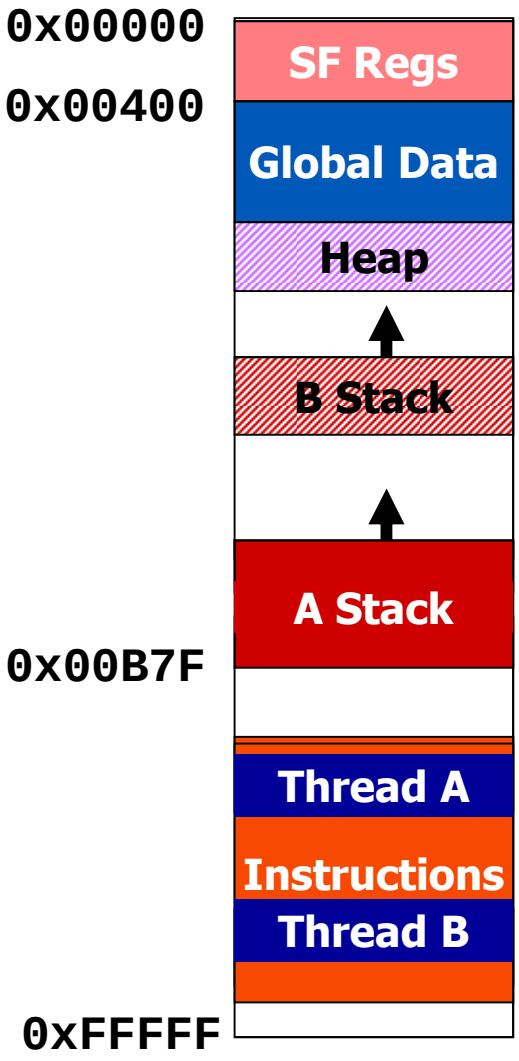
Stack Pointer Monitor

What makes the stack grow?

- Nested subroutine calls – each adds 5 bytes (3 bytes for return address, 2 bytes for dynamic link)
  - Local data in the subroutine call – automatic variables
  - Arguments passed to the subroutine
- Nested interrupt handling – each adds 4 bytes (3 bytes for return address, 1 byte for flag register)
  - Local storage for the interrupt

How large does the stack get?

- Starts at 0x00B7F (top of RAM), grows to smaller addresses
- Will overwrite heap or global data if gets too large
- Need to allocate space for multiple stacks in system with a preemptive scheduler
- Renesas Tool Manager provides some info in asm listing and Stack Viewer



Stack Pointer Monitoring Code

"Solution"

- Examine SP periodically. If SP is below the allowable minimum (SP_LIMIT), reset the system or run a debug routine
- Not guaranteed to detect all stack overflows, but lets us detect some.

Mechanism

- Enhance the Timer B0 overflow interrupt to examine ISP
  - Use stc (Store Control register) instruction and asm macro to store ISP value to variable tmp_SP on stack frame (referenced from Frame Base register FB)

```
#define SP_LIMIT (0x0431)
void tick_timer_intr(void) {
    unsigned int tmp_SP;
    asm(" stc ISP,$$[FB]",
          tmp_SP);
    if (tmp_SP<SP_LIMIT)
          asm("jmp start");
    }
```

- If SP is too small, do something
  - Reset system by jumping to system initialization code (at start)
  - Or start executing a debug routine. However, *there may not be enough space on the stack* to push the debug routine's activation record. May be able to use jump, inline code into the ISR, etc.
- Setting SP_LIMIT
  - Start with beginning of RAM (0x00400)
  - Add in size of globals and possibly heap
  - Increase by some value for a greater margin of safety

Stack Pointer Sampling Code

Useful during system development

- How much space needs to be allocated for the stack?
- Especially useful for multi-tasking systems (multiple stacks)
- What's the cheapest MCU we can buy? (RAM costs money)

Modified "Solution"

- Sample SP periodically. If smaller than minimum value observed so far, save in global variable min_obs_SP
- Not guaranteed to detect minimum stack size, but lets us detect common ones.

Mechanism

- Initialize min_obs_SP to value larger than expected, so first valid access will update it
- Use ISR as before, but update min_obs_SP if needed rather than reset system

```
unsigned int min_obs_SP=0xffff;

void tick_timer_intr(void) {
    unsigned int tmp_SP;
    asm(" stc ISP,$$[FB]",
            tmp_SP);
    if (tmp_SP<min_obs_SP)
            min_obs_SP = tmp_SP;
    }
```

Issues to Consider

Need all ISRs to reenable interrupts to allow TimerB0 ISR to run

This code is statistical, not absolute. It uses sampling to try to find the minimum, but is not guaranteed.

- How long do we need to run the sampling code to have a good sense that we have captured a minimum close to the real minimum?
- Want to make sure code is running in a wide variety of situations – including with many frequent interrupts

How often does this code sample the SP?

- Timer B0 will overflow every 65536/20MHz = 3.2768 ms, or 305.2 times per second

What's the duration of the most-deeply-nested subroutine?

- Might be missed if it's very short.

Mechanisms for robust embedded systems

Watchdog timer

Stack-pointer monitor

Brown-out detector

Brown-Out Detector

Black-out == total loss of electricity

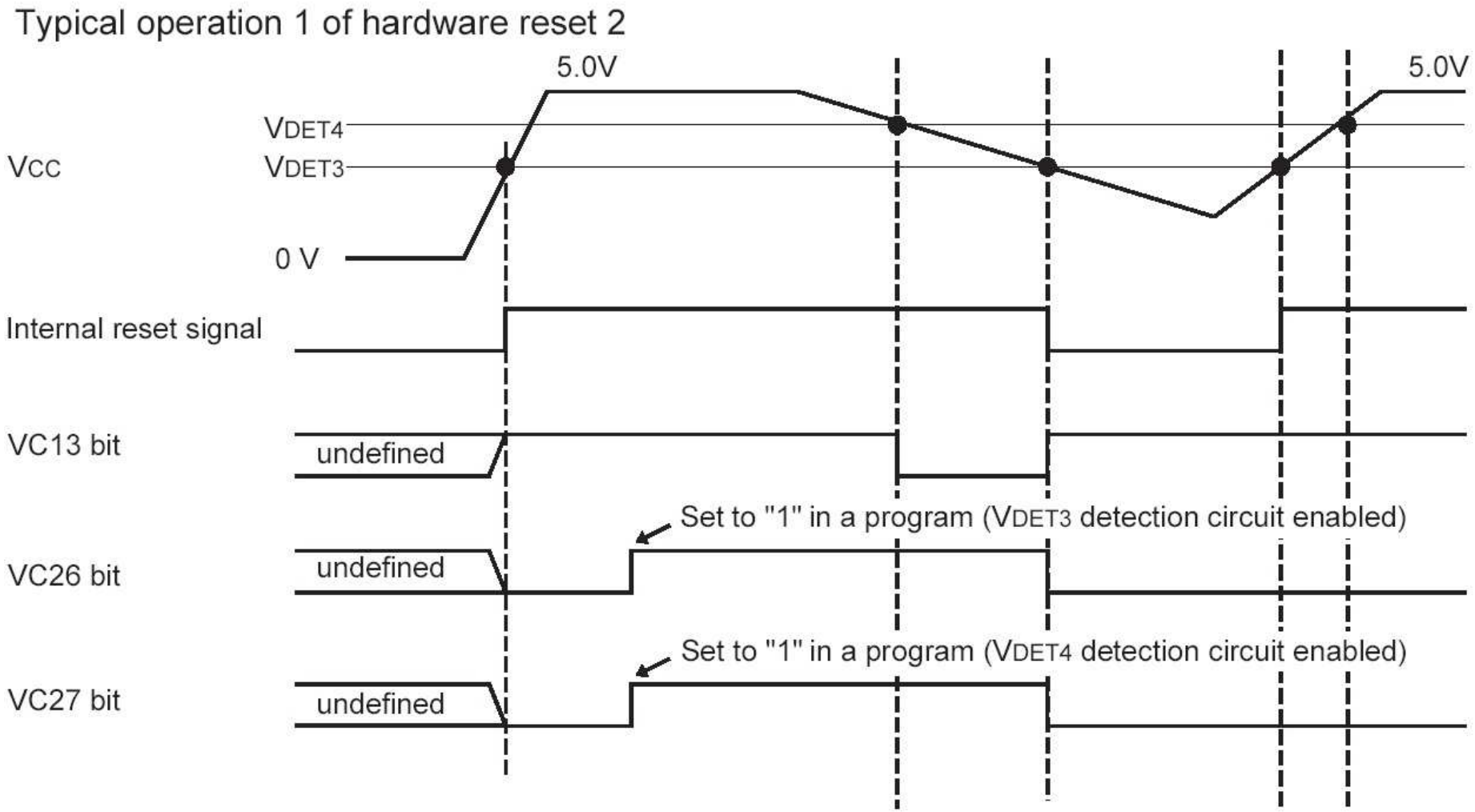
Brown-out == partial loss of electricity

- Voltage is low enough that the system is not guaranteed to work completely
- We can't guarantee that it won't do anything at all. Parts may still work.
  - "CPU runs, except for when trying to do multiplies"

Want to detect brown-out automatically

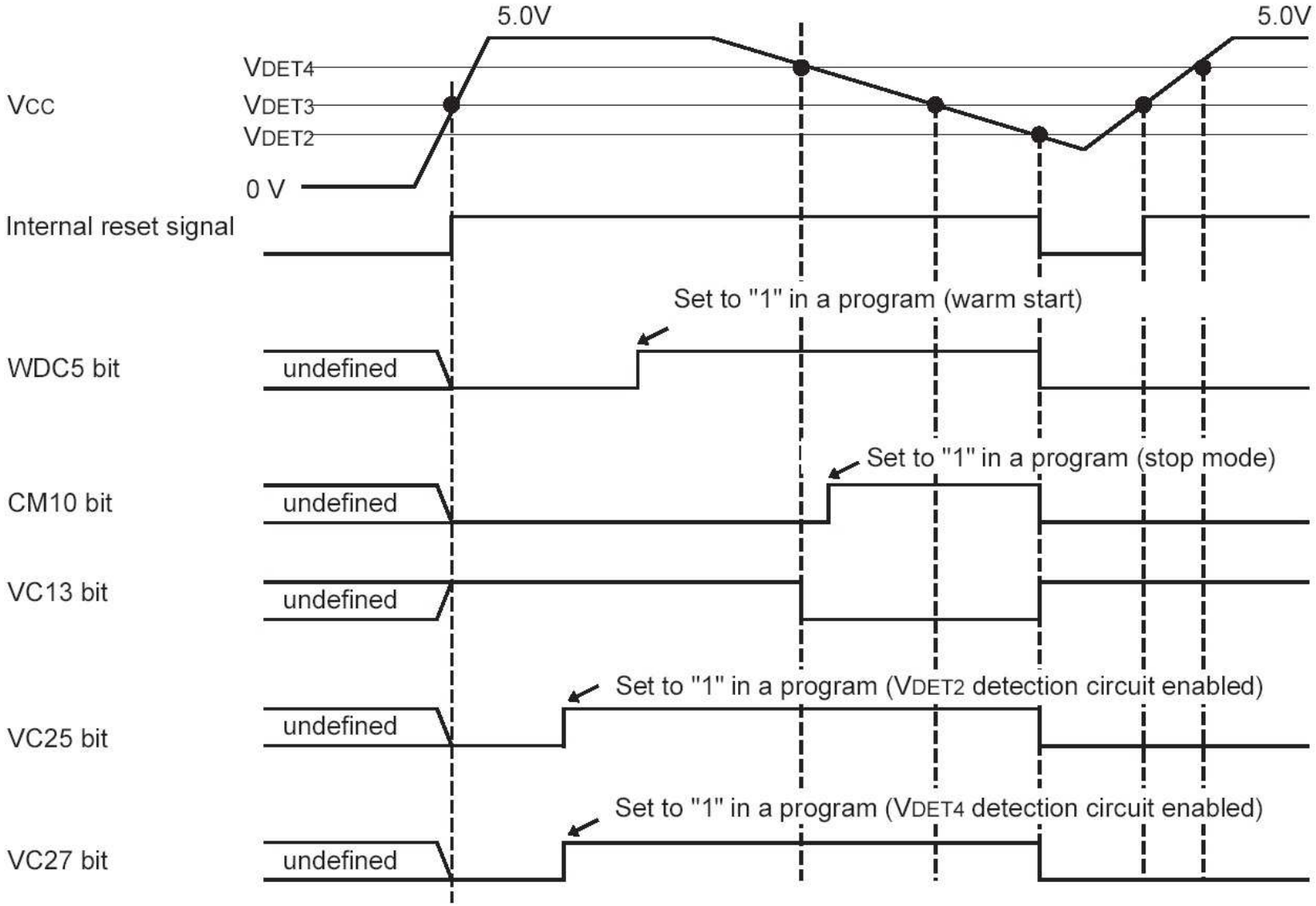
- Possibly save critical processor information to allow warm boot
- Then hold processor in reset state until brown-out ends

M16C/262 Voltage Detector Operation (I)

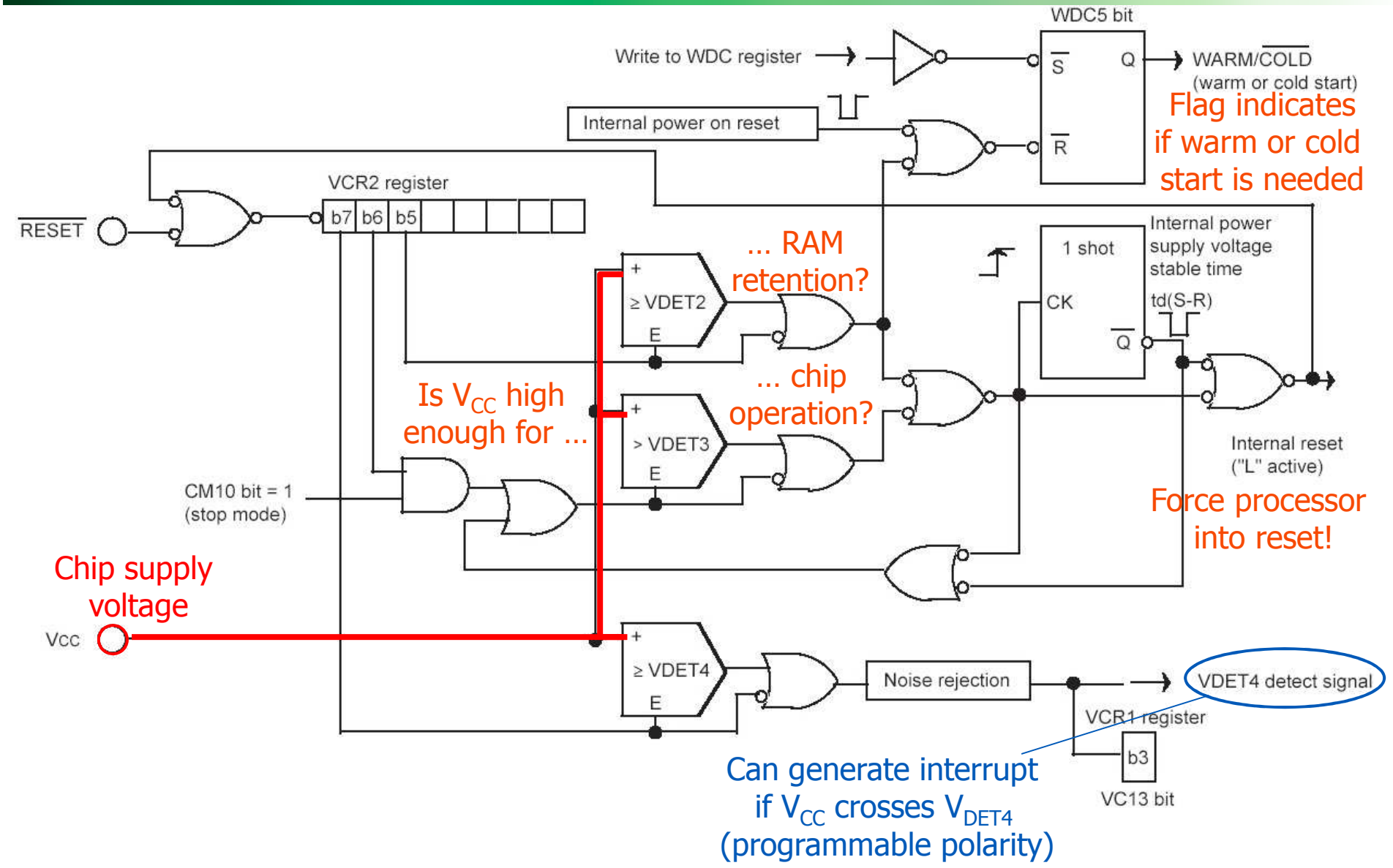


M16C/262 Voltage Detector Operation (II)

Typical operation 2 of hardware reset 2



M16C/262 Voltage Detection Circuit



How to Use the Voltage Detection Circuit (I)

RAM can remember at 2V, rest of the chip needs 2.7 V

- Can choose to reset CPU when V_{CC} falls to V_{DET2} or V_{DET3}

Warm start vs. cold start

- Warm start
  - RAM is guaranteed not to have changed during brown-out
  - => Don't need to reinitialize memory and restart program
    - Remember what ncrt30.asm does? Erases bss section, copies data from ROM to data section, initializes stack pointer, etc. *SLOW*
  - => can instead resume at saved value of PC
- Cold start
  - RAM may have changed (been corrupted) during brown-out, so activation records, data may be corrupted
  - => Need to restart program after reinitializing memory
    - Do this by jumping to the reset vector target (start in our case)

How to Use the Voltage Detection Circuit (II)

VDET4

- Configure to generate interrupt on ***falling edge*** to indicate imminent power failure
  - Put system into safe mode (stop motors, turn off laser, etc.)
  - Save critical data in non-volatile memory if available
- Configure to generate interrupt on ***rising edge*** to start up processor after going into stop mode (to save power)
- Shares interrupt vector with watchdog timer and stopped oscillator detector