

Module 10. Operating Modes and More

Topics

- Revisiting interrupts
- Operating modes

ADC12 Interrupts

Our examples so far have used the ADC12SC bit to start conversions, and then we poll the ADC12BUSY bit to see when a conversion is complete:

LED_ON() ← DEMO: USE I/O PIN (LED) TO MEASURE TIME
 WE WAIT FOR ADC

```

ADC12CTL0 |= ADC12SC;

LED_OFF()
while (ADC12CTL1 & ADC12BUSY) {
    __no_operation(); // Could just leave body of loop empty
}
LED_ON()
in_value = ADC12MEM0 & 0x0FFF;
  
```

CONVERSION TAKES $\approx 200\mu s$

This is another form of *busy-waiting*, which is like `swDelay`:

- The CPU isn't doing any useful work—it's just sitting in a loop!
- While ADC conversions happen very quickly ($\ll 1$ ms), the CPU executes faster, so we do need to wait for a result

The main purpose of on-chip peripherals like the Timer and ADC is to remove burdens from or provide services to the CPU.

→ TIMERS, ADC, ETC OPERATE INDEPENDENTLY
FROM THE CPU!

↑
 (ASYNCHRONOUSLY)

The solution to this is to use an **Interrupt**, an external signal sent by a peripheral requesting that the CPU do something. Interrupts are hard-wired into certain peripherals. Fortunately for us, the ADC12 can trigger interrupts.

On the ADC12, an interrupt can signal the end of a conversion, **meaning that it is ready for the CPU to read data from its memory registers.**

Here is an example configuration with two channels:

```

ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12ON | ADC12MSC;
ADC12CTL1 = ADC12SHP | ADC12CONSEQ_1;

// Here, we are performing conversions for two channels
ADC12MCTL0 = ADC12SREF_0 + ADC12INCH_5;
ADC12MCTL1 = ADC12SREF_1 + ADC12INCH_6 + ADC12EOS;

// Because we are converting for two channels, we want the interrupt
// to occur after BOTH conversions are complete, so we enable the
// interrupt for MEM1.
ADC12IE = BIT1;
    TRIGGER INT AFTER CH. 1
    ↑
    ←
__enable_interrupt(); // Globally enable interrupts

ADC12CTL0 |= ADC12SC + ADC12ENC; // Enable ADC and start conversion

```

What should the ISR do? It's triggered when a conversion is finished, so it just needs to read the memory registers!

```

// Global variables for storing data
// (could also store into an array!)
volatile unsigned int in_value1, in_value2;

#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
    // Move the results for both channels into global variables
    in_value1 = ADC12MEM0 & 0x0FFF;
    in_value2 = ADC12MEM1 & 0x0FFF;
}

```

CONVERSION IS DONE HERE

“Scheduling” ADC measurements

It's also possible to have the ADC perform conversions automatically. An easier option is to trigger ADC conversions from a timer and use ADC interrupts to read the results.

```
// NOTE: this example assumes the timer and ADC have already been configured.

volatile unsigned int in_value1, in_value2;

// Timer A2 ISR
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2_ISR(void)
{
    timer++; // KEEP TIME AS USUAL

    ADC12CTL0 |= ADC12SC; ← START CONVERSION
}

// ADC 12 ISR
#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
    in_value1 = ADC12MEM0 & 0x0FFF;
    // . . .
}

void main(void)
{
    setup_everything();
    _enable_interrupt();

    while(1){
        display_adc_values(in_value1, in_value2);
    }
}
```

Here, we now have two ISRs, one for the ADC, and one for the timer. Thus, our main program can simply use `in_value1` and `in_value2` without needing to explicitly start conversions.

↳ REMOVES SOME BURDEN FROM MAIN().

BUT CAN WE DO BETTER?

Polling versus Interrupts Revisited

So far most of our code has relied heavily on Polling.

-- We've been using Timer A2 as a time base, but our main functions are still does busy-waiting in some form of loop:

```
while (1)
{
  if (global_time_cnt > last_time)
  {
    take_ADC_meas();
    last_time = global_time_count;
  }
  button = checkButtons();
  . . .

  // Other task(s) can occur at different intervals
  if ((global_time_cnt % cnt_per_second) == 0)
  {
    toggleLED(LED2)
    displayHHMMSS(global_time_cnt);
  }
  . . .
}
```

TASK:
TASK: READ ADC

TASK:
UPDATE
DISPLAY

As an alternative, we can organize our code to *schedule* tasks to occur at specific “real” times, and even assign priorities to tasks.

--> We do this by implementing the Scheduler INSIDE Timer ISR!

```
// A simple task scheduler for the MSP430F5529
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2(void)
{
    // First priority: maintain time base
    global_time_cnt++;

    // Some of the app task(s) may execute every time slice
    take_ADC_meas();
    button = checkButtons();
    . . .

    // Other application task(s) at different intervals
    if ((global_time_cnt % cnt_per_second) == 0)
    {
        toggleLED(LED2)
        displayHHMMSS(global_time_cnt);
    }
    . . . .
}

```

DIFFERENT
TASKS
"SCHEDULED"
REGULARLY
BY TIMER

THESE ~~APP~~ TASKS
ONLY RUN WHEN
ISR FIRES.
NO BUSY WAITING!

Here, main() would consist of initialization followed by an “empty” loop:

```
void main()
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    init_sys(); // Initialize the MSP430
    setupButton(); // configure buttons
    setupADC(); // configure ADC

    . . .

    _enable_interrupt(); // Global Interrupt enable
    runTimerA2(); // Start scheduler

    // An empty forever loop!
    // All application tasks are scheduled and dispatched
    // from within TimerA2 ISR
    while(1)
    {
        __no_operation();
    }
}

```

MAIN() DOES NOTHING!!
ALL WORK HANDLED BY ISR.

GOAL: REPLACE THIS WITH SLEEP().
SAVE ENERGY!

What are we assuming by organizing or scheduling our application's tasks like this?

There some very important assumptions here:

- ISR MUST BE DONE BEFORE
NEXT X_{INT}

- ALL ~~TASKS~~ TASKS MUST BE DONE WITHIN ONE X_{INT} .

- CODE IN ISR MUST NOT "BLOCK" (WAIT)
FOR LONGER THAN X_{INT}

Remember the rule of interrupts!

- We can have multiple interrupt sources for different peripherals—we want to maintain this behavior when scheduling tasks.
- By default, interrupts are disabled inside an ISR. Therefore, one interrupt cannot normally interrupt another one, but we can change this behavior when we need it.

- CAN STILL HAVE MULTIPLE INTERRUPT
SOURCES

- NORMALLY CAN'T HAVE ONE INTERRUPT
INTERRUPT ANOTHER, BUT CAN CHANGE THIS.

- THIS CAN GET COMPLEX - CAN USE TO
IMPLEMENT MULTITASKING.

- WANT TO KNOW MORE? TRY ECE 3849

"REAL-TIME OS"

INTERRUPTS HAVE FIXED PRIORITIES AND MAY PRE-EMPT ONE ANOTHER.

Interrupt Priorities

Interrupt Vector Addresses

The interrupt vectors and the power-up start address are located in the address range 0FFFFh to 0FF80h. The vector contains the 16-bit address of the appropriate interrupt-handler instruction sequence.

FIXED PRIORITIES

Table 4. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power-Up External Reset Watchdog Timeout, Password Violation Flash Memory Password Violation	WDTIFG, KEYV (SYSRSTIV) ⁽¹⁾⁽²⁾	Reset	0FFFEh	63, highest
System NMI PMM Vacant Memory Access JTAG Mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV) ⁽¹⁾	(Non)maskable	0FFFCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV) ⁽¹⁾⁽²⁾	(Non)maskable	0FFFAh	61
Comp_B	Comparator B interrupt flags (CBIV) ⁽¹⁾⁽³⁾	Maskable	0FFF8h	60
TB0	TB0CCR0 CCIFG0 ⁽³⁾	Maskable	0FFF6h	59
TB0	TB0CCR1 CCIFG1 to TB0CCR6 CCIFG6, TB0IFG (TB0IV) ⁽¹⁾⁽³⁾	Maskable	0FFF4h	58
Watchdog Timer_A Interval Timer Mode	WDTIFG	Maskable	0FFF2h	57
USCI_A0 Receive or Transmit	UCA0RXIFG, UCA0TXIFG (UCA0IV) ⁽¹⁾⁽³⁾	Maskable	0FFF0h	56
USCI_B0 Receive or Transmit	UCB0RXIFG, UCB0TXIFG (UCB0IV) ⁽¹⁾⁽³⁾	Maskable	0FFEEh	55
ADC12_A	ADC12IFG0 to ADC12IFG15 (ADC12IV) ⁽¹⁾⁽³⁾⁽⁴⁾	Maskable	0FFECCh	54
TA0	TA0CCR0 CCIFG0 ⁽³⁾	Maskable	0FFEAh	53
TA0	TA0CCR1 CCIFG1 to TA0CCR4 CCIFG4, TA0IFG (TA0IV) ⁽¹⁾⁽³⁾	Maskable	0FFE8h	52
USB_UBM	USB interrupts (USBIV) ⁽¹⁾⁽³⁾	Maskable	0FFE6h	51
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV) ⁽¹⁾⁽³⁾	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0 ⁽³⁾	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV) ⁽¹⁾⁽³⁾	Maskable	0FFE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0 ⁽³⁾	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV) ⁽¹⁾⁽³⁾	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV) ⁽¹⁾⁽³⁾	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV) ⁽¹⁾⁽³⁾	Maskable	0FFD2h	41
Reserved	Reserved ⁽⁵⁾		0FFD0h	40
			⋮	⋮
			0FF80h	0, lowest

UNMASKABLE

HIGHER PRIORITY

ADC12_A

TA2

(1) Multiple source flags
 (2) A reset is generated if the CPU tries to fetch instructions from within peripheral space or vacant memory space.
 (Non)maskable: the individual interrupt-enable bit can disable an interrupt event, but the general-interrupt enable cannot disable it.
 (3) Interrupt flags are located in the module.
 (4) Only on devices with ADC, otherwise reserved.
 (5) Reserved interrupt vectors at addresses are not used in this device and can be used for regular program code if necessary. To maintain compatibility with other devices, it is recommended to reserve these locations.

Operating Modes

What's the advantage of scheduling tasks?

>>>Our CPU doesn't need to be running all the time!!

This means we can use the MSP430's operating modes to save energy when not performing tasks.

The MSP430F5529 has 6 Operating Modes:

WE ARE HERE!

Active Mode => "Normally Active" = CPU is active, all enabled clocks are active

Low Power Mode 0 => CPU, MCLK are disabled, SMCLK, ACLK are active

↳ LPM0

Low Power Mode 1 => CPU, MCLK, DCO osc. are disabled, DC generator is disabled if the DCO is not used for MCLK or SMCLK in active mode, SMCLK, ACLK are active

Low Power Mode 2 => CPU, MCLK, SMCLK, DCO osc. are disabled, DC generator remains enabled, ACLK is active

Low Power Mode 3 => CPU, MCLK, SMCLK, DCO osc. are disabled, DC generator disabled, ACLK is active

Low Power Mode 4 => CPU and all clocks disabled (RAM retention mode)

Low Power Mode 4.5 => CPU and all clocks disabled (no RAM retention), PWR management off, Digital IO pin configuration retained

LPM4.5

MORE FUNCTIONS DISABLED

WAKE TIME INCREASES

POWER USAGE DECREASES

RAM RETENTION: RAM IS VOLATILE, SO CAN POWER OFF RAM, BUT WILL LOSE ALL DATA STORED THERE. SHOULD !!

Operating Mode Basics

AAA BATTERY IS RATED w/
CAPACITY OF 1100 mA.h

>> Most embedded systems run on batteries

-- Enter Low Power Mode to conserve batteries

ACTIVE MODE @ 1MHz, 3V

$$\frac{1100 \text{ mA}\cdot\text{h}}{0.36 \text{ mA}} = 3055 \text{ HOURS}$$

I_{AM, TYPICAL} = ~~0.36~~ 0.36 mA

≈ 127 DAYS

On MSP430F5529 Low Power Modes are indicated/selected thru the CPUOFF, OSCOFF, SCG0, and SCG1 bits in the Status Register (SR)

WHAT IF WE GO TO LPM3 FOR 50% OF THE TIME?

Mode	SCG0	SCG1	OSCOFF	CPUOFF
Active	0	0	0	0
LPM0	0	0	0	1
LPM1	0	1	0	1
LPM2	1	0	0	1
LPM3	1	1	0	1
LPM4	1	1	1	1
LPM4.5	1	1	1	1

I_{TYP, LPM3} = 2.1 μA

$$(0.5)(0.36 \text{ mA}) + (0.5)(2.1 \mu\text{A})$$

I_{AVG} = 181 μA AVERAGE

$$\frac{1100 \text{ mA}\cdot\text{h}}{181 \mu\text{A}} = 6058 \text{ HOURS}$$

ALMOST DOUBLED BATTERY LIFE! → ≈ 253 DAYS!

*PMMREGOFF bit = 1, as well

Status Register Layout:

Reserved	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C
----------	------	------	--------	--------	-----	---	---	---

How do you enter a Low Power Mode?

Can set with the `_BIS_SR()` function:

```
Ex:  _BIS_SR(LPM3_bits) ; // set bits to enter LMP3
```

```
    _BIS_SR(LPM3_bits|GIE) ; // set bits to enter LMP3
                                // and enable interrupts
```

↑ SET BITS IN STATUS REGISTER.

When to enter LPM and how do you exit?

--> Ideally want to enter LPM whenever not executing tasks

--> This is made simple if program is organized as a "Scheduler"

↳ **EVENT DRIVEN PROGRAMMING.**

--> Enter LPM after starting timer in main()

```
// A simple task scheduler for the MSP430F5529
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2(void)
{
    // First priority = maintain the time base
    global_time_cnt++;

    // Some app task(s) may execute every time slice
    take_ADC_meas();
    button =checkButtons();
    . . .

    // Other application task(s) execute less frequently
    if ((global_time_cnt % cnt_per_second) == 0)
        displayHHMMSS();
    . . . . .
}
}
```

How do you exit LPM if CPU is OFF?

• >> **INTERRUPTS!**

When an interrupt is received from a certain source, the CPU automatically does the following:

1. Finishes its current (assembly) instruction
2. Saves Status Register (SR) and Program Counter (PC) to stack
3. Clears Status Register (set to 0) ← **GOES INTO ACTIVE MODE!**
4. Loads address of ISR that was triggered from Interrupt Vector Table (IVT), loads it into PC

5. Execution continues in ISR

6. ← **WHEN ISR IS DONE, PREVIOUS PC, SR ARE RESTORED**

Returning from an ISR restores the Status Register to its previous values meaning that your program will automatically return to Low Power Mode when exiting the ISR!

Power MONITORING: $V_{CC} = 3V, 1MHz, 25^{\circ}C$

10A



MSP430F5529, MSP430F5528, MSP430F5527, MSP430F5526
 MSP430F5525, MSP430F5524, MSP430F5522, MSP430F5521
 MSP430F5519, MSP430F5517, MSP430F5515, MSP430F5514, MSP430F5513

www.ti.com

SLAS590L - MARCH 2009 - REVISED MAY 2013

Low-Power Mode Supply Currents (Into V_{CC}) Excluding External Current

over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted)^{(1) (2)}

PARAMETER	V_{CC}	PMMCOREVx	-40°C		25°C		60°C		85°C		UNIT
			TYP	MAX	TYP	MAX	TYP	MAX	TYP	MAX	
$I_{LPM0,1MHz}$ Low-power mode 0 ⁽³⁾⁽⁴⁾	2.2 V	0	73		77	85	80		85	97	μA
	3.0 V	3	79		83	92	88		95	105	
I_{LPM2} Low-power mode 2 ⁽⁵⁾⁽⁴⁾	2.2 V	0	6.5		6.5	12	10		11	17	μA
	3.0 V	3	7.0		7.0	13	11		12	18	
$I_{LPM3,XT1LF}$ Low-power mode 3, crystal mode ⁽⁶⁾⁽⁴⁾	2.2 V	0	1.60		1.90		2.6		5.6		μA
		1	1.65		2.00		2.7		5.9		
		2	1.75		2.15		2.9		6.1		
	3.0 V	0	1.8		2.1	2.9	2.8		5.8	8.3	
		1	1.9		2.3		2.9		6.1		
		2	2.0		2.4		3.0		6.3		
		3	2.0		2.5	3.9	3.1		6.4	9.3	
$I_{LPM3,VLO}$ Low-power mode 3, VLO mode ⁽⁷⁾⁽⁴⁾	3.0 V	0	1.1		1.4	2.7	1.9		4.9	7.4	μA
		1	1.1		1.4		2.0		5.2		
		2	1.2		1.5		2.1		5.3		
		3	1.3		1.6	3.0	2.2		5.4	8.5	
I_{LPM4} Low-power mode 4 ⁽⁸⁾⁽⁴⁾	3.0 V	0	0.9		1.1	1.5	1.8		4.8	7.3	μA
		1	1.1		1.2		2.0		5.1		
		2	1.2		1.2		2.1		5.2		
		3	1.3		1.3	1.6	2.2		5.3	8.1	
$I_{LPM4.5}$ Low-power mode 4.5 ⁽⁹⁾	3.0 V		0.15		0.18	0.35	0.26		0.5	1.0	μA

Electrical Characteristics

Active Mode Supply Current Into V_{CC} Excluding External Current

over recommended operating free-air temperature (unless otherwise noted)^{(1) (2) (3)}

PARAMETER	EXECUTION MEMORY	V_{CC}	PMMCOREVx	FREQUENCY ($f_{DCO} = f_{MCLK} = f_{SMCLK}$)										UNIT	
				1 MHz		8 MHz		12 MHz		20 MHz		25 MHz			
				TYP	MAX	TYP	MAX	TYP	MAX	TYP	MAX	TYP	MAX	TYP	MAX
$I_{AM, Flash}$	Flash	3.0 V	0	0.36	0.47	2.32	2.60								mA
			1	0.40		2.65		4.0	4.4						
			2	0.44		2.90		4.3		7.1	7.7				
			3	0.46		3.10		4.6		7.6		10.1	11.0		
$I_{AM, RAM}$	RAM	3.0 V	0	0.20	0.24	1.20	1.30							mA	
			1	0.22		1.35		2.0	2.2						
			2	0.24		1.50		2.2		3.7	4.2				
			3	0.26		1.60		2.4		3.9		5.3	6.2		

- (1) All inputs are tied to 0 V or to V_{CC} . Outputs do not source or sink any current.
- (2) The currents are characterized with a Micro Crystal MS1V-T1K crystal with a load capacitance of 12.5 pF. The internal and external load capacitance are chosen to closely match the required 12.5 pF.
- (3) Characterized with program executing typical data processing. USB disabled ($V_{USBEN} = 0$, $SLDOEN = 0$).
 $f_{ACLK} = 32786$ Hz, $f_{DCO} = f_{MCLK} = f_{SMCLK}$ at specified frequency.
 $XTS = CPUOFF = SCG0 = SCG1 = OSCOFF = SMCLKOFF = 0$.


```

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    init_sys();    // initialize system
    _BIS_SR(GIE);

    . . .

    runtimerA2(); // task scheduler runs in Timer A2 ISR

    . . .
    _BIS_SR(LPM0_bits|GIE); // Enter low power mode
    // Arrival of timer interrupt will cause MSP430 to exit low
    // power mode and enable it to execute all tasks within the
    // Timer A2 ISR

    while(1) {
        // Main program loop does nothing!
    }
}

```

Which LPM you enter depends on which clocks you are using for your scheduling Timer.

-- Also, on how quickly you need other clocks

- **DEPENDING ON MODE, CLOCKS WILL BE DISABLED**

~~***~~ - **LPM2-LPM4.5 - SMCLK IS OFF**

SO COULD STOP TIMER, IF USING SMCLK.

IN LPM4, ALL CLOCKS ARE OFF, BUT YOU CAN WAKE UP W/ AN I/O EVENT (LIKE A BUTTON)!

>> Low Power Modes are great in practice, but can make debugging painful!

-- It's simple so add it last when debugging is complete

-- Then test to see if '430 is waking and executing tasks properly

Computing Power Usage: Example

An MSP430F5529 is being powered by a Duracell 2/3A LiMnO₂ battery, which has a nominal voltage of 3V and a listed capacity of 1550 mAh.

The MSP430F5529 is running an application that is in Active Mode for 5.6% of the time, in ~~PM4 for 10.7% of the time~~, and in LPM4 for the rest of the time.

How long can the application run using this battery?

ASSUME 1MHz, 25°C, TYPICAL VALUES

① STATE ASSUMPTIONS →

- ACTIVE MODE 5.6% $I_{AM} = 0.36 \text{ mA}$

- LPM4 1-5.6% $I_{LPM4} = 1.1 \mu\text{A}$

② FIND AVERAGE CURRENT

DANGER: MIXED UNITS!!

$$I_{AVG} = (0.056)(0.36 \text{ mA}) + (1-0.056)(1.1 \mu\text{A})$$

$$I_{AVG} = 21.198 \mu\text{A}$$

③ FIND TIME

$$\frac{1550 \text{ mAh}}{21.198 \mu\text{A}} = 73118 \text{ HRS}$$

$$\approx 3046 \text{ DAYS}$$

$$\approx \boxed{8 \text{ YEARS}}$$

!!
O

ADC PROBLEM

16-BIT ADC

$$V_{REF+} = 1.8V, V_{REF-} = 0V$$

d. FSR

$$FSR = V_{REF+} - V_{REF-} = 1.8V - 0V = \boxed{1.8V}$$

d. RESOLUTION = ~~FSR~~ $\frac{FSR}{2^k} = \frac{1.8V}{2^{16}} = \boxed{27.466 \mu V/BIT}$

e. CODE = ~~0x4051~~ $0x4051$, $V_{IN}?$

$$CODE = 16465$$

$$CODE = \text{FLOOR} \left(\frac{V_{IN} - V_{REF-}}{V_{REF+} - V_{REF-}} \right) (2^k - 1)$$

$$16465 = \frac{V_{IN}}{1.8V} (2^{16} - 1)$$

$$\boxed{V_{IN} = 452.23 mV}$$

d. $V_{IN} = 0.92V$

CODE = ?

$$CODE = \left\lceil \left(\frac{V_{IN} - V_{REF-}}{V_{REF+} - V_{REF-}} \right) (2^k - 1) \right\rceil$$

↑
Floor

$$= \frac{0.92V}{1.8V} (2^{16} - 1) = \lfloor 33495.67 \rfloor$$

$$= \boxed{33495}$$

CODE MUST BE AN INTEGER!