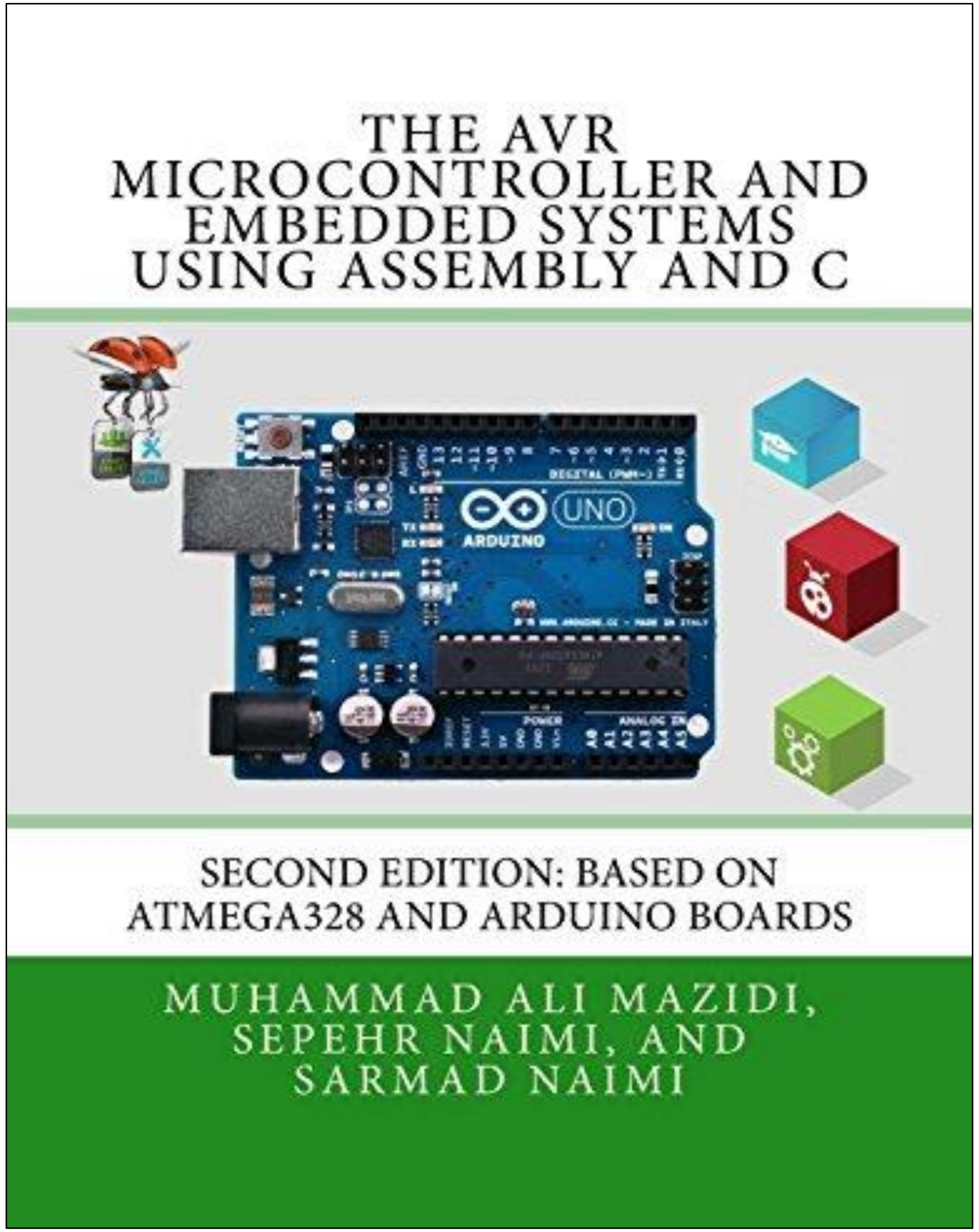

THE AVR MICROCONTROLLER AND EMBEDDED SYSTEMS USING ASSEMBLY AND C

SECOND EDITION: BASED ON ATMEGA328 AND ARDUINO BOARDS

MUHAMMAD ALI MAZIDI, SEPEHR NAIMI, AND SARMAD NAIMI

LECTURE # 05M: System Clock Options

Prof. Dr. Engr. Muhibul Haque Bhuyan
Professor, Department of EEE
American International University-Bangladesh (AIUB)
Dhaka, Bangladesh

Introduction:

Oscillating Options:

Every microcontroller needs a clock source. The CPU, the memory bus, the peripherals—clock signals are everywhere inside a microcontroller. They govern the speed at which the processor executes instructions, the baud rate of serial communication signals, the amount of time needed to perform an analog-to-digital conversion, and so much more.

All these clocking actions return to the clock signal's source, namely the oscillator. Therefore, the oscillator must provide the desired performance from the microcontroller. Though some oscillator options are more complex or expensive than others, the choice of oscillator should reflect the importance of reducing cost and complexity whenever possible. **CPU clock speed is a good indicator of the processor's performance**. In general, a higher clock speed means a faster CPU. A CPU with a clock speed of 3.2 GHz executes 3.2 billion cycles per second.

Introduction:

There are quite a few ways to generate a clock signal for a microcontroller. The datasheet of the microcontroller should provide a good deal of information about what types of oscillators can be used and how to implement them in a way that is compatible with its hardware. There are some advantages and disadvantages of various clock sources and thus it is needed to choose the best one among the oscillator options available for a particular microcontroller. There are **two broad types of oscillators-** internal and external.

Internal

A resistor-capacitor circuit (R-C phase shift oscillator circuit)
Phase-Locked Loop (PLL) for frequency multiplication

External

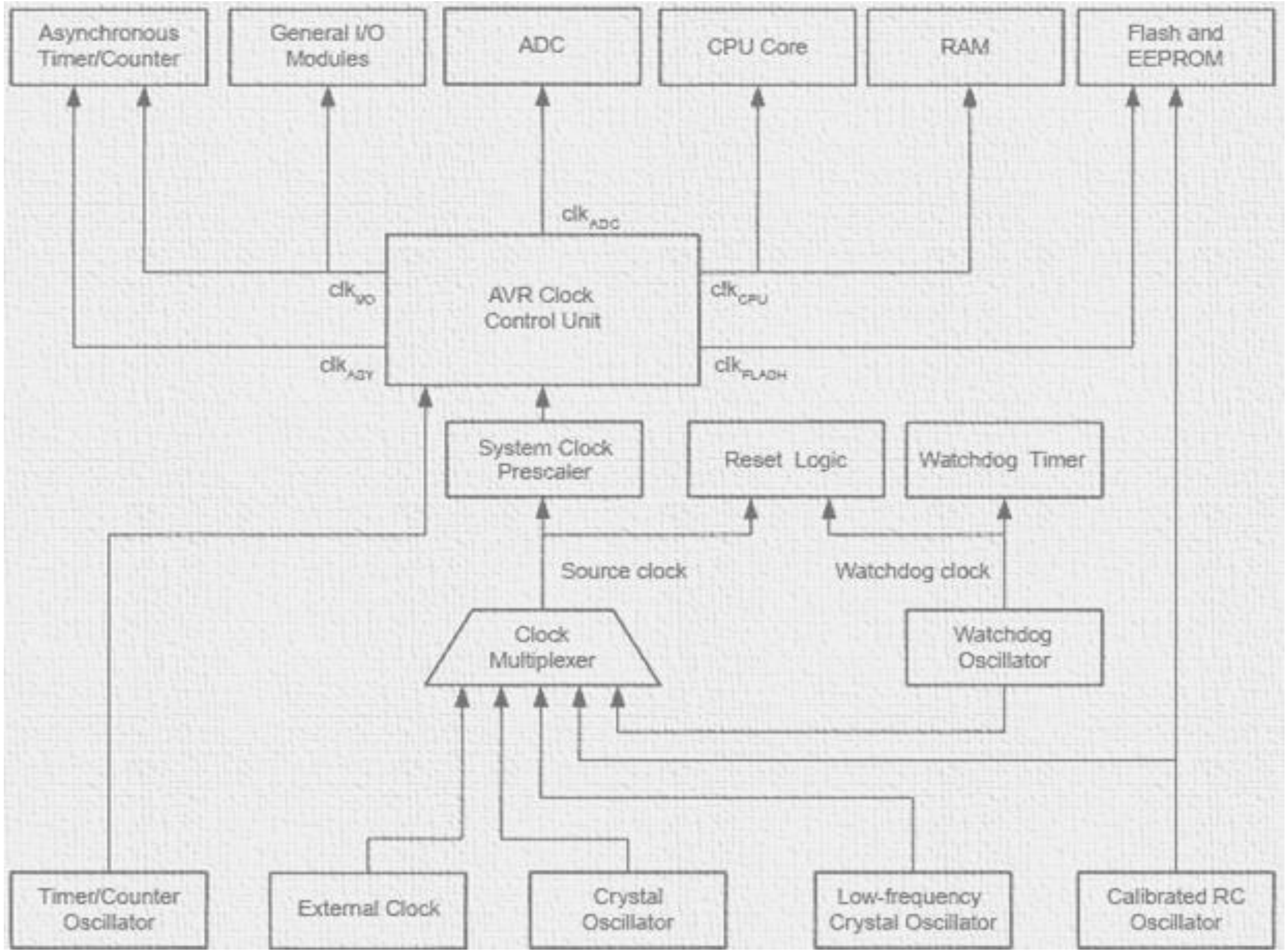
CMOS clock, crystal oscillator, ceramic resonator oscillator, resistor-capacitor oscillator, capacitor-only oscillator, etc.

Block Diagram:

The block diagram represents the **principal clock systems** in the AVR and their distribution.

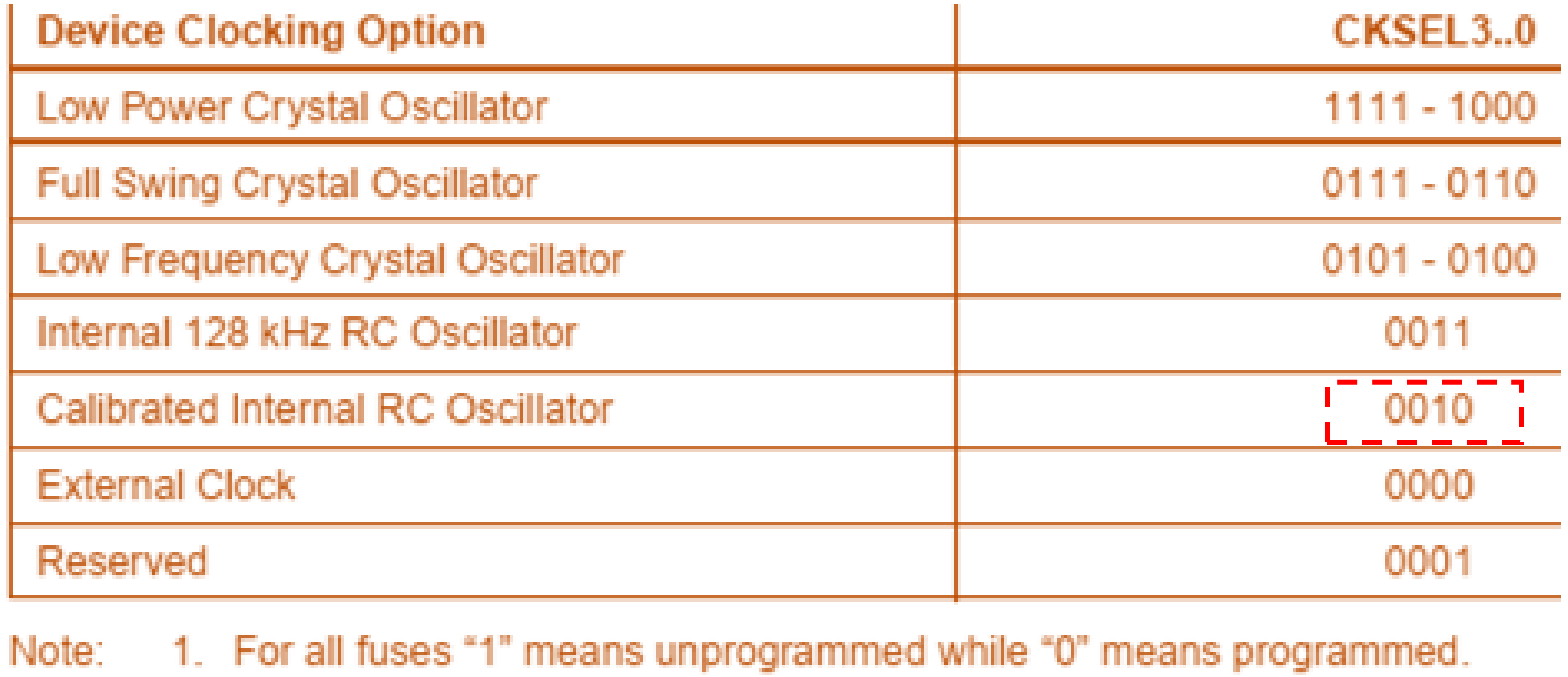
All the clocks need not be active at a given time.

To reduce power consumption, the clocks to modules not being used can be **halted** by using different **sleep modes**.



Clock Sources:

The device has the following clock source options, selectable by **Flash Fuse bits** as shown. The clock from the selected source is input to the **AVR clock generator** and routed to the appropriate modules



Device Clocking Option	CKSEL3..0
Low Power Crystal Oscillator	1111 - 1000
Full Swing Crystal Oscillator	0111 - 0110
Low Frequency Crystal Oscillator	0101 - 0100
Internal 128 kHz RC Oscillator	0011
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved	0001

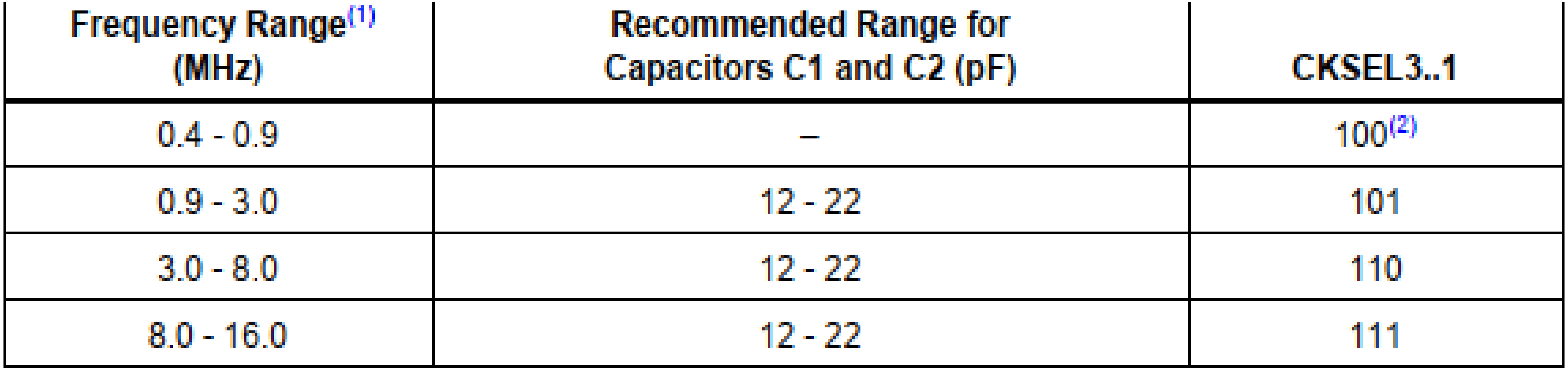
Note: 1. For all fuses "1" means unprogrammed while "0" means programmed.

Default clock source:

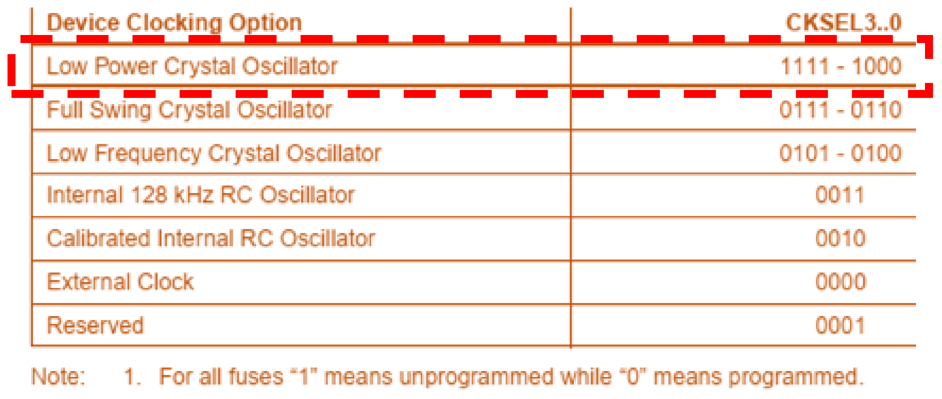
The device is shipped with an internal RC oscillator at 8 MHz and the fuse CKDIV8 programmed, resulting in a 1 MHz system clock.
The start-up time (SUT) is set to maximum, and the time-out period is enabled (CKSEL = "0010", SUT = "10", CKDIV8 = "0"). The default setting ensures that all users can make their desired clock source setting using any available programming interface.
The term “overclocking” refers to speeding up the CPU clock for more processing power. Intel CPUs with a “K” in the name have an unlocked “multiplier” for easy overclocking when paired with a motherboard chipset that supports overclocking.
Overclocking can yield **improved FPS**, even for high-end CPUs like the latest **Intel® Core™ i9 processor**.

Low Power Crystal Oscillator:

Pins XTAL1 and XTAL2 are input and output respectively of an inverting amplifier, which can be configured for use as an on-chip oscillator.



Frequency Range[1] (MHz)	Recommended Range for Capacitors C1 and C2 (pF)	CKSEL3..1
0.4 - 0.9	–	100[2]
0.9 - 3.0	12 - 22	101
3.0 - 8.0	12 - 22	110
8.0 - 16.0	12 - 22	111

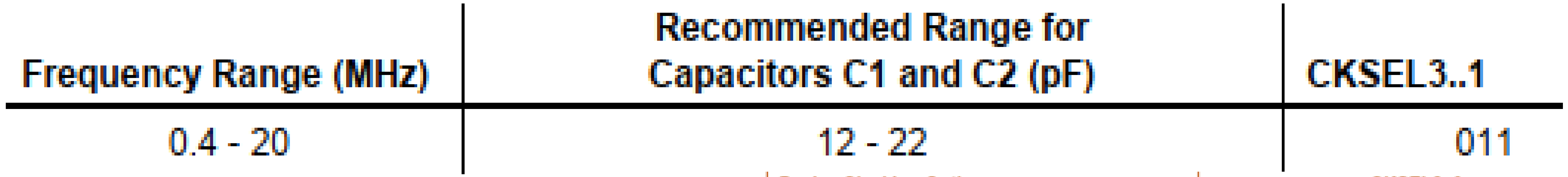


Device Clocking Option	CKSEL3..0
Low Power Crystal Oscillator	1111 - 1000
Full Swing Crystal Oscillator	0111 - 0110
Low Frequency Crystal Oscillator	0101 - 0100
Internal 128 kHz RC Oscillator	0011
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved	0001

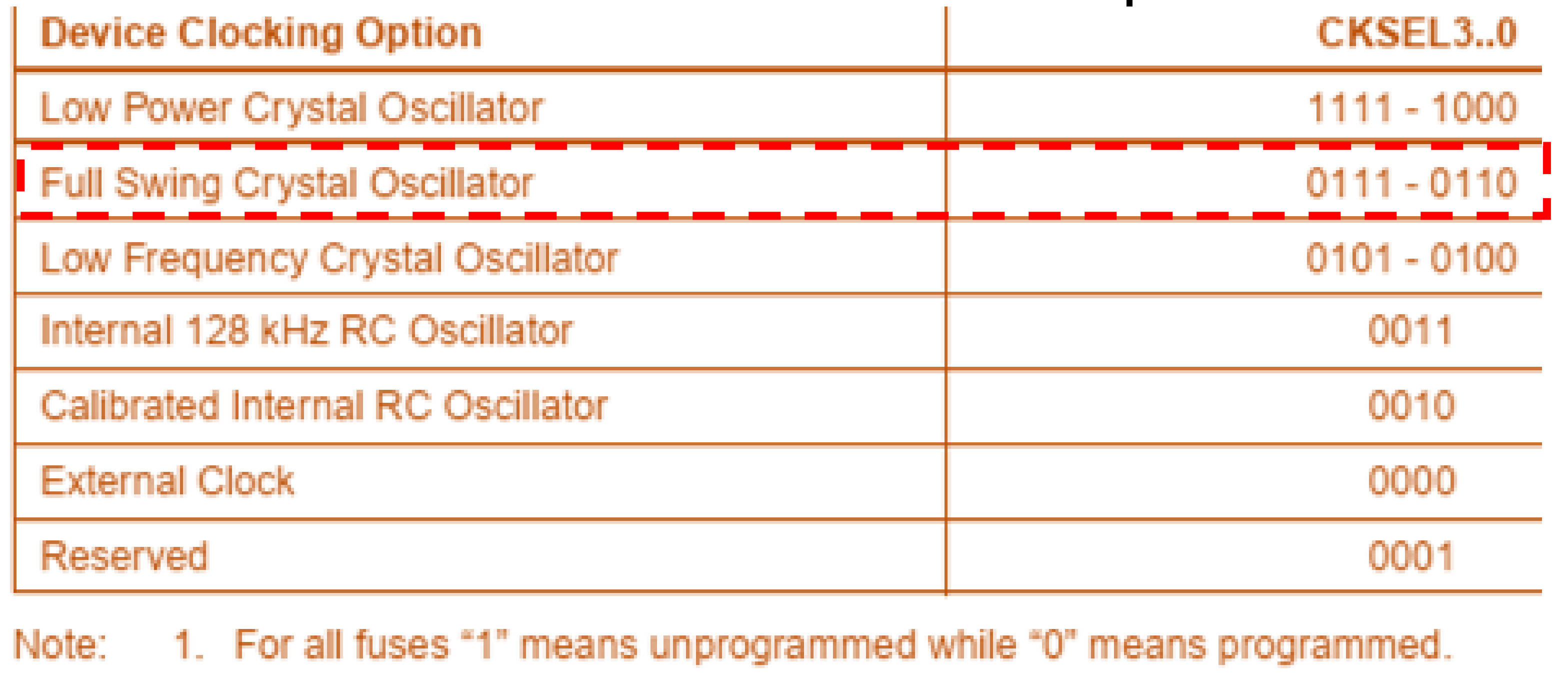
Note: 1. For all fuses "1" means unprogrammed while "0" means programmed.

Full swing Crystal Oscillator:

This crystal oscillator is a full swing oscillator, with a rail-to-rail swing on the XTAL2 output. This is useful for driving other clock inputs in a noisy environment.



Frequency Range (MHz)	Recommended Range for Capacitors C1 and C2 (pF)	CKSEL3..1
0.4 - 20	12 - 22	011



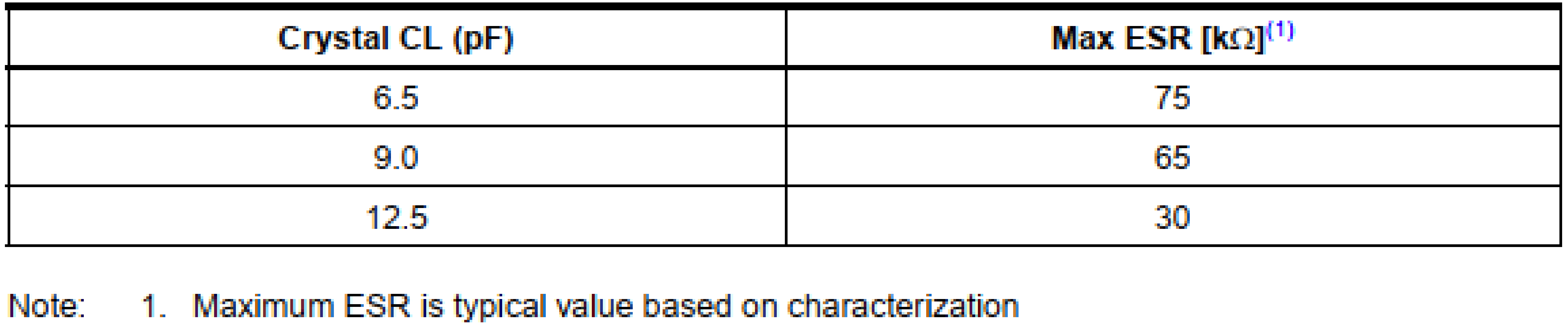
Device Clocking Option	CKSEL3..0
Low Power Crystal Oscillator	1111 - 1000
Full Swing Crystal Oscillator	0111 - 0110
Low Frequency Crystal Oscillator	0101 - 0100
Internal 128 kHz RC Oscillator	0011
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved	0001

Note: 1. For all fuses "1" means unprogrammed while "0" means programmed.

Low frequency Crystal Oscillator:

The **low-frequency crystal** oscillator is optimized for use with a **32.768 kHz watch crystal**. When selecting crystals, the load capacitance and the crystal’s **Equivalent Series Resistance (ESR)** must be taken into consideration. Both values are specified by the crystal vendor.

ATmega48P/88P/168P/328P oscillator is optimized for very low power consumption, and thus when selecting crystals, for maximum ESR, recommended load capacitances, C_L are 6.5 pF, 9.0 pF, and 12.5 pF for crystals



Crystal CL (pF)	Max ESR [kΩ][1]
6.5	75
9.0	65
12.5	30

Note: 1. Maximum ESR is typical value based on characterization

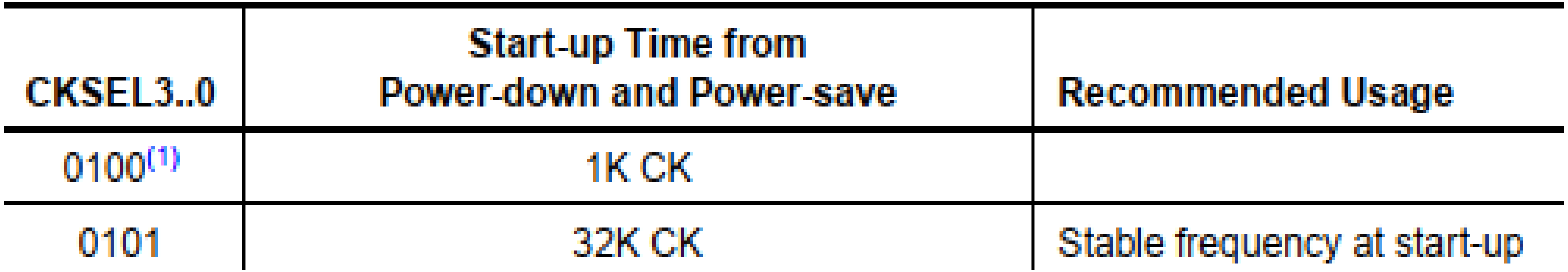
Note: 1. This option should only be used if frequency stability at start-up is not important for the application

Low-Frequency Crystal Oscillator:

The low-frequency crystal oscillator provides an **internal load capacitance of a typical 6 pF** at each **TOSC pin**. The **external capacitance (C)** needed at each TOSC pin can be calculated by using:

$C = 2*C_L - C_S$ where C_L is the load capacitance for a **32.768 kHz crystal** specified by the crystal vendor and C_S is the total stray capacitance for one TOSC pin.

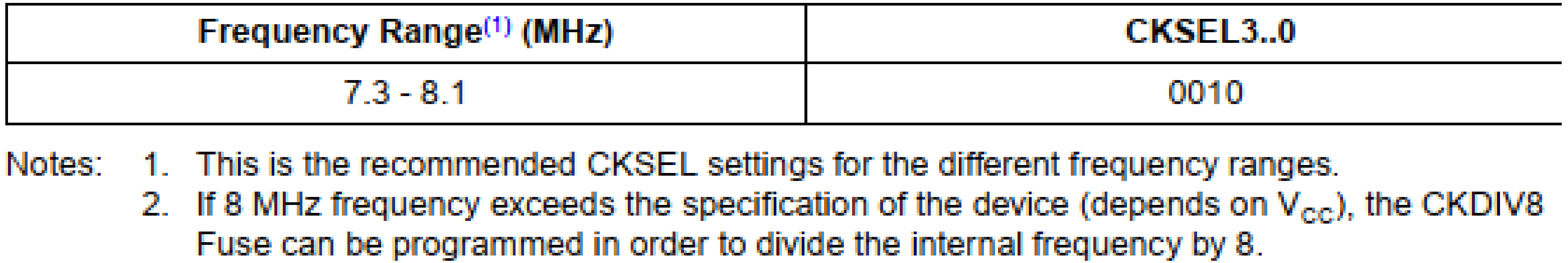
Crystals specifying load capacitance (C_L) higher than 6 pF, require external capacitors to be applied. The low-frequency crystal oscillator must be selected by setting the CKSEL fuses to "0110" or "0111". Start-up times are determined by the SUT fuses.



CKSEL3..0	Start-up Time from Power-down and Power-save	Recommended Usage
0100[1]	1K CK	
0101	32K CK	Stable frequency at start-up

Calibrated Internal RC Oscillator:

By default, the internal RC oscillator provides an approximate **8 MHz clock**. Though voltage and temperature dependent, this clock can be very accurately calibrated by the user. The device is shipped with the **CKDIV8 Fuse programmed**. This clock may be selected as the system clock by **programming the CKSEL Fuses**.



Frequency Range[1] (MHz)	CKSEL3..0
7.3 - 8.1	0010

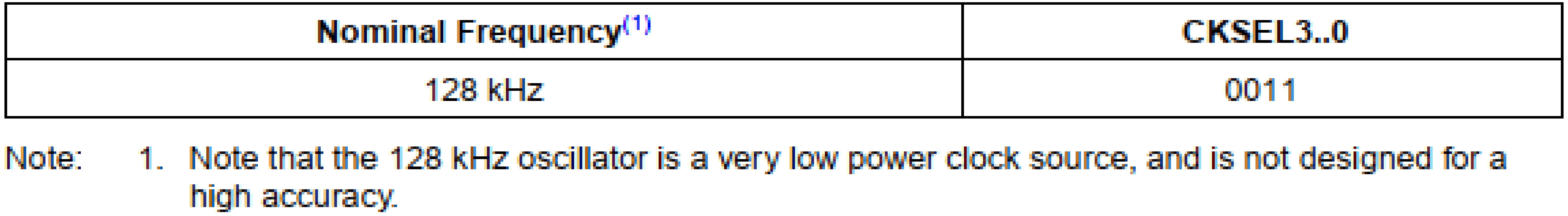
Notes: 1. This is the recommended CKSEL settings for the different frequency ranges.
2. If 8 MHz frequency exceeds the specification of the device (depends on V_{CC}), the CKDIV8 Fuse can be programmed in order to divide the internal frequency by 8.

When this Oscillator is selected, start-up times are determined by the SUT Fuses.

If selected, it will operate with no external components. During reset, the hardware loads the pre-programmed calibration value into the **OSCCAL Register** and thereby automatically calibrates the RC oscillator.

128 kHz Internal Oscillator:

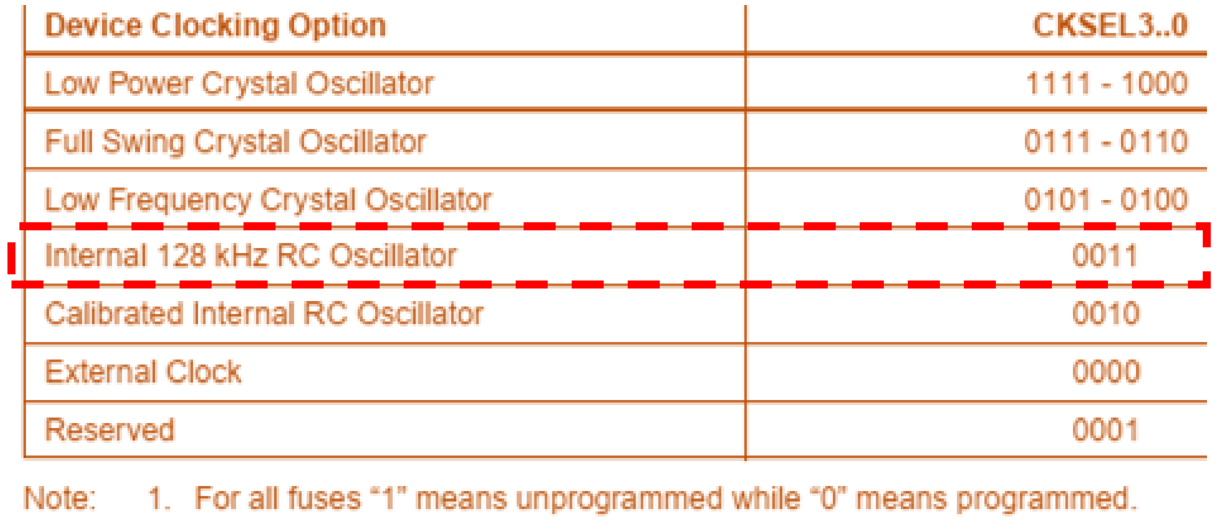
The 128 kHz internal oscillator is a low-power oscillator providing a clock of 128 kHz. The frequency is nominal at 3 V and 25°C. This clock may be selected as the system clock by programming the CKSEL Fuses to “11”.



Nominal Frequency[1]	CKSEL3..0
128 kHz	0011

Note: 1. Note that the 128 kHz oscillator is a very low power clock source, and is not designed for a high accuracy.

When this clock source is selected, **start-up times are determined by the SUT Fuses**.

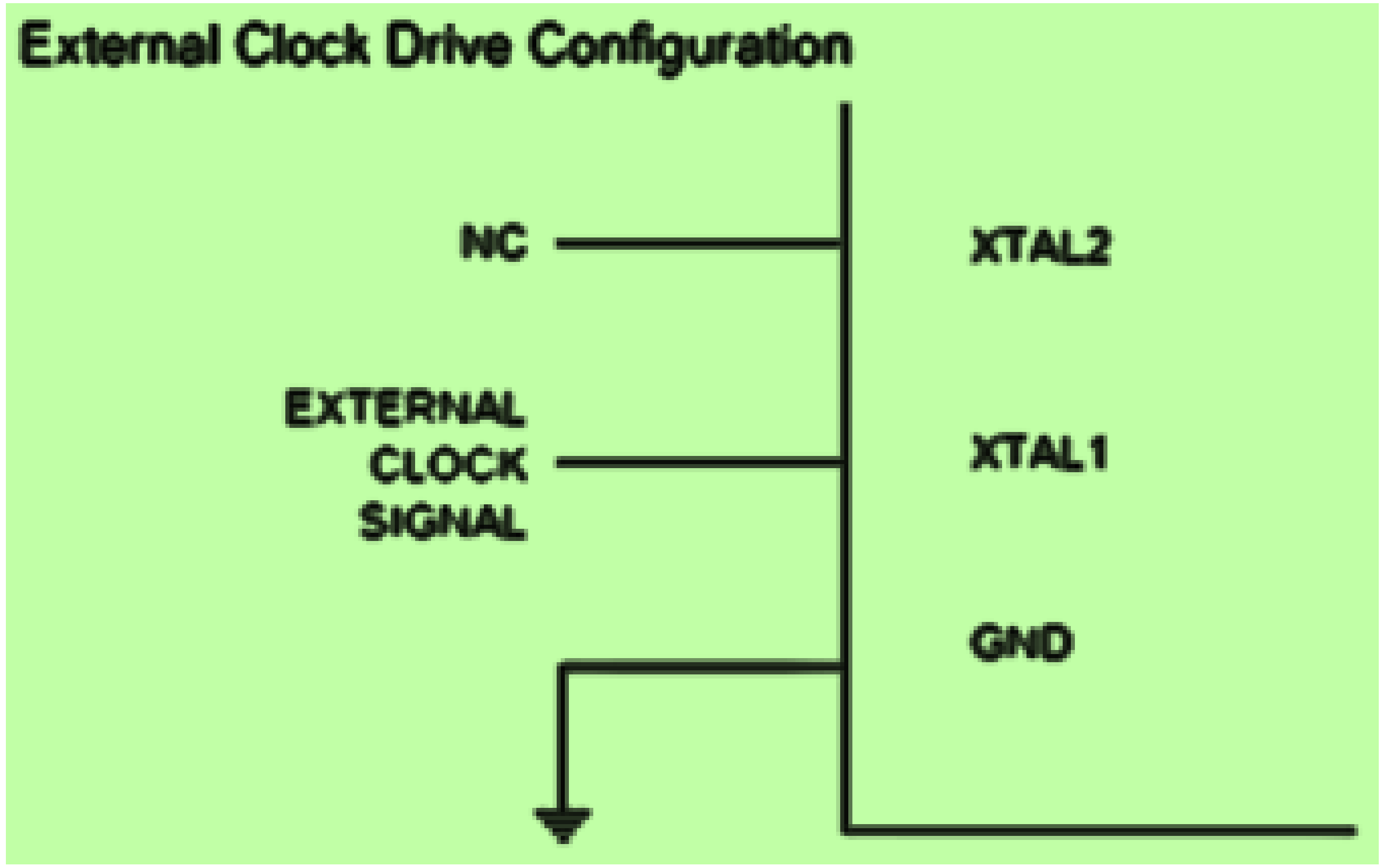


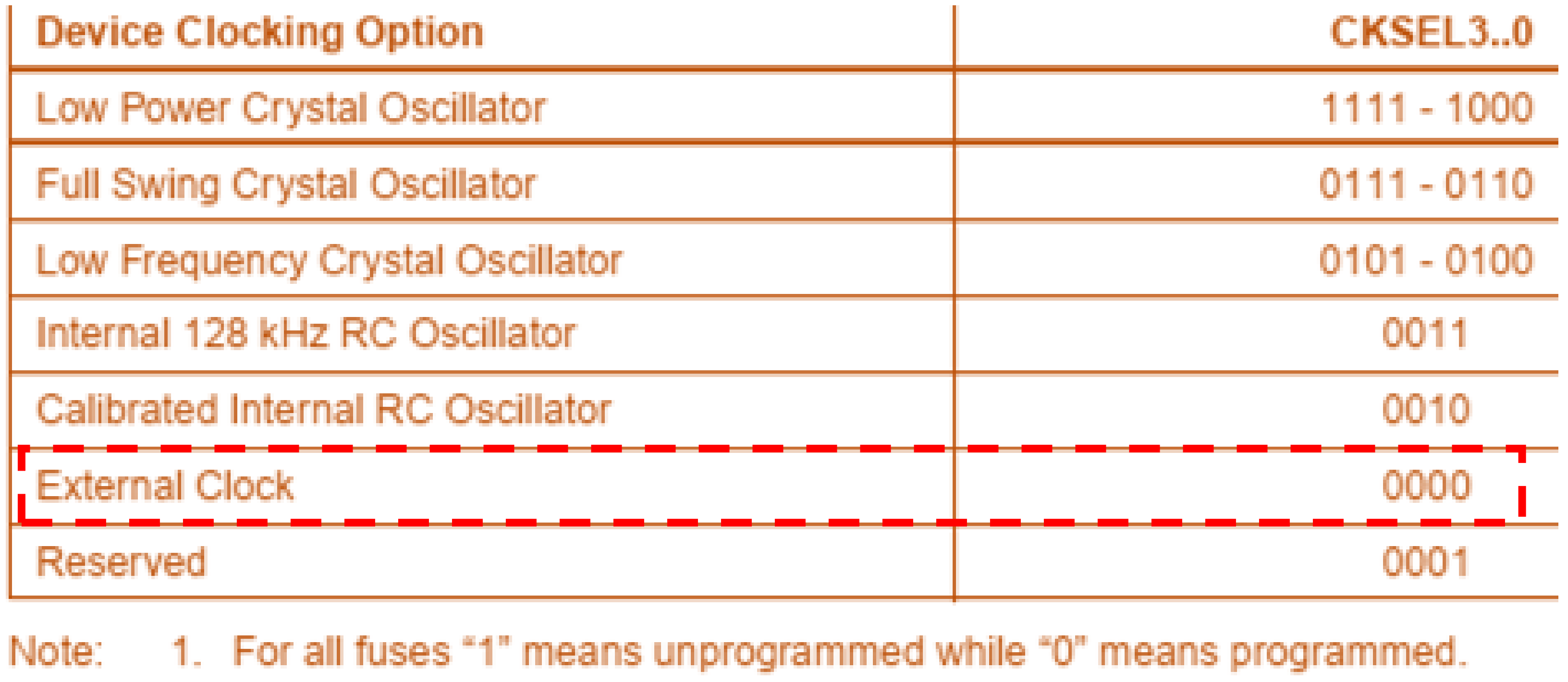
Device Clocking Option	CKSEL3..0
Low Power Crystal Oscillator	1111 - 1000
Full Swing Crystal Oscillator	0111 - 0110
Low Frequency Crystal Oscillator	0101 - 0100
Internal 128 kHz RC Oscillator	0011
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved	0001

Note: 1. For all fuses “1” means unprogrammed while “0” means programmed.

External Clock:

To drive the device from an external clock source, XTAL1 should be driven as follows-





Device Clocking Option	CKSEL3..0
Low Power Crystal Oscillator	1111 - 1000
Full Swing Crystal Oscillator	0111 - 0110
Low Frequency Crystal Oscillator	0101 - 0100
Internal 128 kHz RC Oscillator	0011
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved	0001

Note: 1. For all fuses "1" means unprogrammed while "0" means programmed.

To run the device on an external clock, the CKSEL Fuses must be programmed to "0000"

Frequency	CKSEL3..0
0 - 20 MHz	0000

Timer/Counter Oscillator:

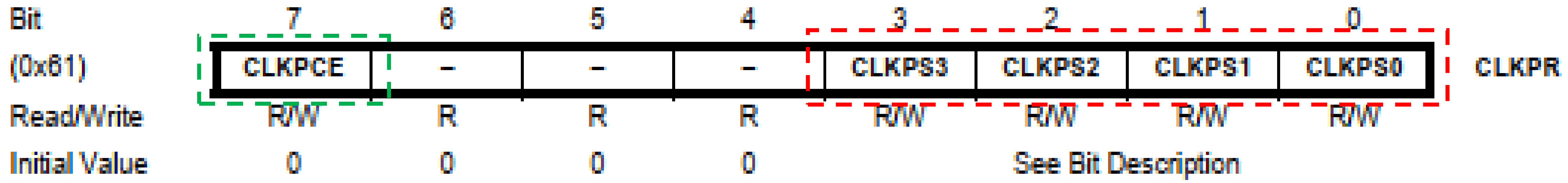
- ATmega48P/88P/168P/328P uses the same crystal oscillator for the Low-frequency oscillator and Timer/Counter oscillator.
- ATmega48P/88P/168P/328P share the **Timer/Counter Oscillator Pins** (**TOSC1 and TOSC2**) with XTAL1 and XTAL2 respectively.
- When using the Timer/Counter Oscillator, the system clock needs to be four times the oscillator frequency. Due to this and the pin sharing, the Timer/Counter Oscillator can only be used when the calibrated internal RC oscillator is selected as the system clock source.
- Applying an external clock source to TOSC1 can be done if EXTCLK in the **ASSR Register** is written to logic one. An external clock is selected as input instead of a **32.768 kHz watch crystal**.

System Clock Pre-Scalar:

- The ATmega48P/88P/168P/328P has a system **clock Pre-scaler**, and the system clock can be divided by setting the ”CLKPR – Clock Pre-scaler Register”.
- This feature can be used to **decrease the system clock frequency and power consumption** when the requirement for processing power is low.
- This can be used with all clock source options, and it will **affect the clock frequency** of the CPU and all synchronous peripherals.

Register Description:

• CLKPR-Clock Pre-scaler Register



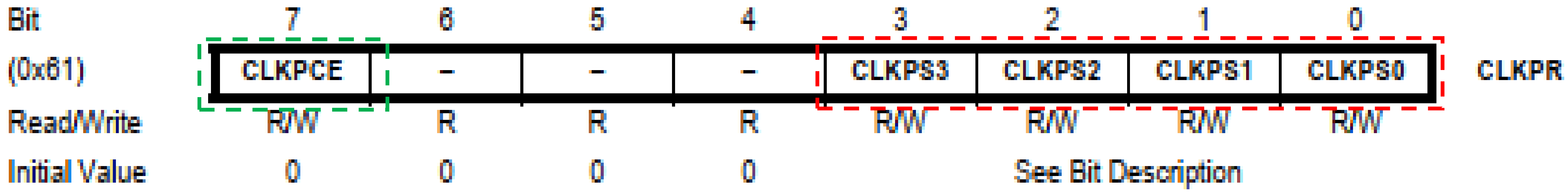
Bit	7	6	5	4	3	2	1	0	
(0x61)	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0	CLKPR
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	See Bit Description				

• Bit 7 – CLKPCE: Clock Pre-scaler Change Enable

The CLKPCE bit must be **written to logic one to enable the change** of the CLKPS bits. The CLKPCE bit is only updated when the other bits in CLKPR are simultaneously written to zero. CLKPCE is **cleared by hardware four cycles after it is written or when CLKPS bits** are written. Rewriting the CLKPCE bit within this time-out period does neither extend the time-out period nor clear the CLKPCE bit.

Register Description:

- CLKPR-Clock Pre-scaler Register

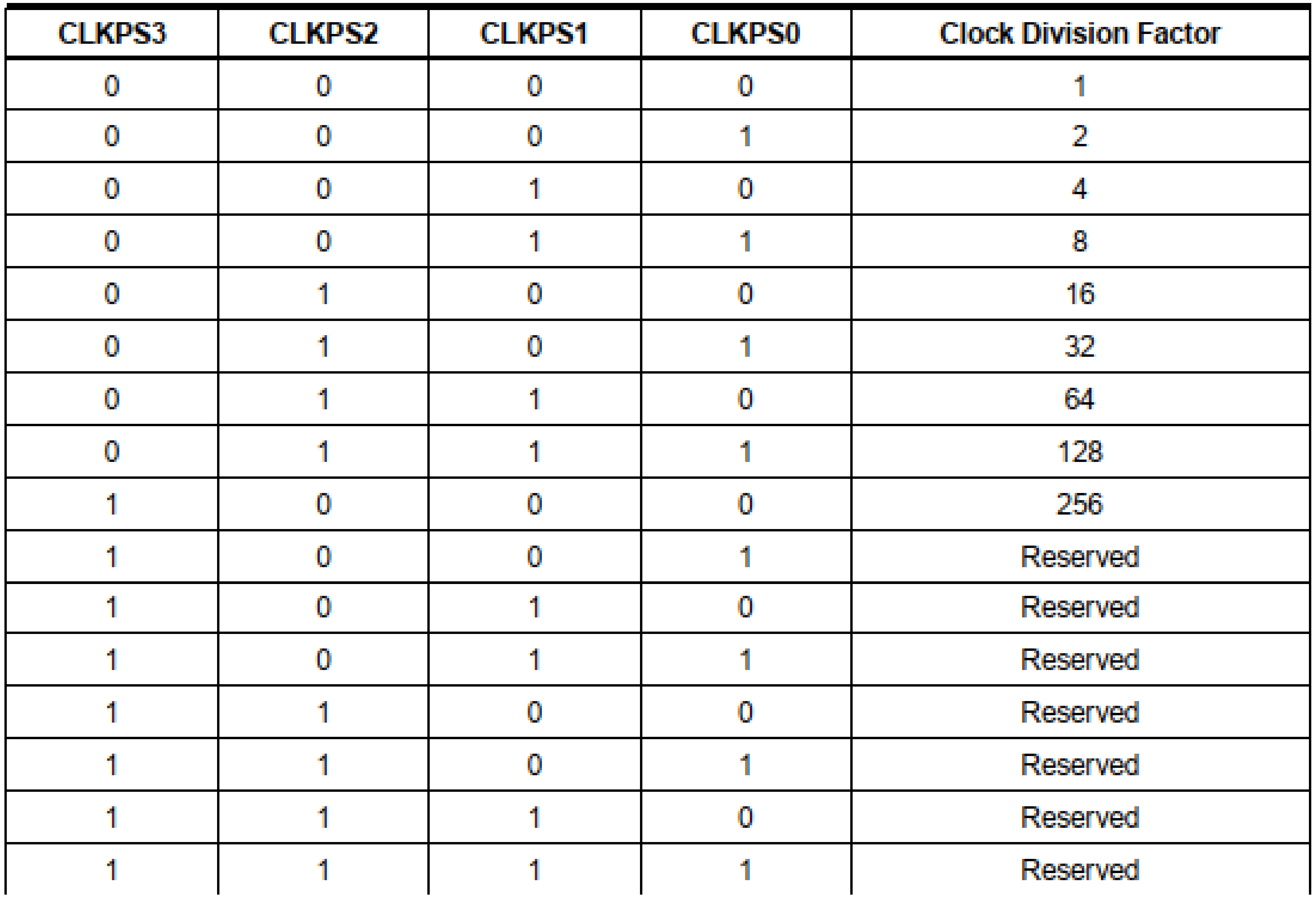


Bit	7	6	5	4	3	2	1	0	
(0x61)	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0	CLKPR
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	See Bit Description				

- Bits 3..0 – CLKPS3..0: Clock Pre-scaler Select Bits 3 - 0

These bits define the **division factor between** the selected clock source and the internal system clock. These bits can be written run-time to vary the clock frequency to suit the application requirements. As the divider divides the **master clock input to the MCU**, the speed of all **synchronous peripherals is reduced** when a division factor is used.

Register Description: Clock Pre-Scaler Select



CLKPS3	CLKPS2	CLKPS1	CLKPS0	Clock Division Factor
0	0	0	0	1
0	0	0	1	2
0	0	1	0	4
0	0	1	1	8
0	1	0	0	16
0	1	0	1	32
0	1	1	0	64
0	1	1	1	128
1	0	0	0	256
1	0	0	1	Reserved
1	0	1	0	Reserved
1	0	1	1	Reserved
1	1	0	0	Reserved
1	1	0	1	Reserved
1	1	1	0	Reserved
1	1	1	1	Reserved

Power Management and Sleep Modes:

Power consumption is a critical issue for a device running continuously for a long time without being turned off. So, to overcome this problem almost every controller comes with a **Sleep Mode**, which helps developers to design electronic gadgets. Sleep mode puts the device in power-saving mode optimal **power consumption** by turning off the unused modules.

An Arduino Sleep mode is also referred to as **Arduino Power Save Mode** or **Arduino Standby Mode**.

Arduino UNO, Arduino Nano, and Pro-mini come with ATmega328P and it has a **Brown-Out Detector (BOD)**, which monitors the supply voltage at the time of **Sleep Mode**.

Power Management and Sleep Modes:

Sleep modes enable the application to shut down unused modules in the MCU, thereby saving power. The AVR provides various sleep modes allowing the user to tailor power consumption to the application’s requirements. There are **six sleep modes** in ATmega328P:

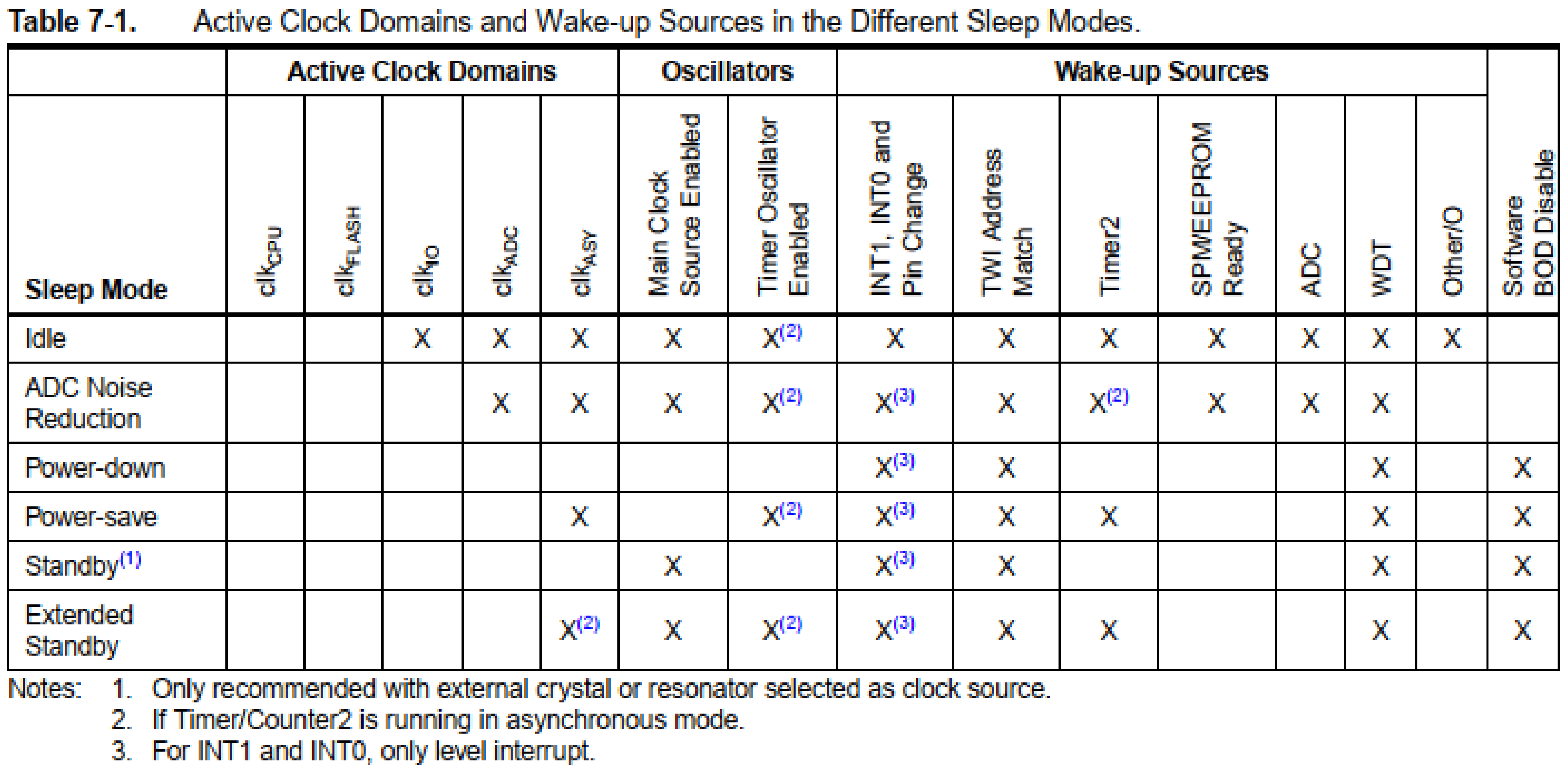


Table 7-1. Active Clock Domains and Wake-up Sources in the Different Sleep Modes.

	Active Clock Domains					Oscillators		Wake-up Sources							
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT1, INT0 and Pin Change	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other/O	Software BOD Disable
Idle			X	X	X	X	X[2]	X	X	X	X	X	X	X	
ADC Noise Reduction				X	X	X	X[2]	X[3]	X	X[2]	X	X	X		
Power-down								X[3]	X				X		X
Power-save					X		X[2]	X[3]	X	X			X		X
Standby[1]						X		X[3]	X				X		X
Extended Standby					X[2]	X	X[2]	X[3]	X	X			X		X

Notes: 1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.
3. For INT1 and INT0, only level interrupt.

Power Management and Sleep Modes:

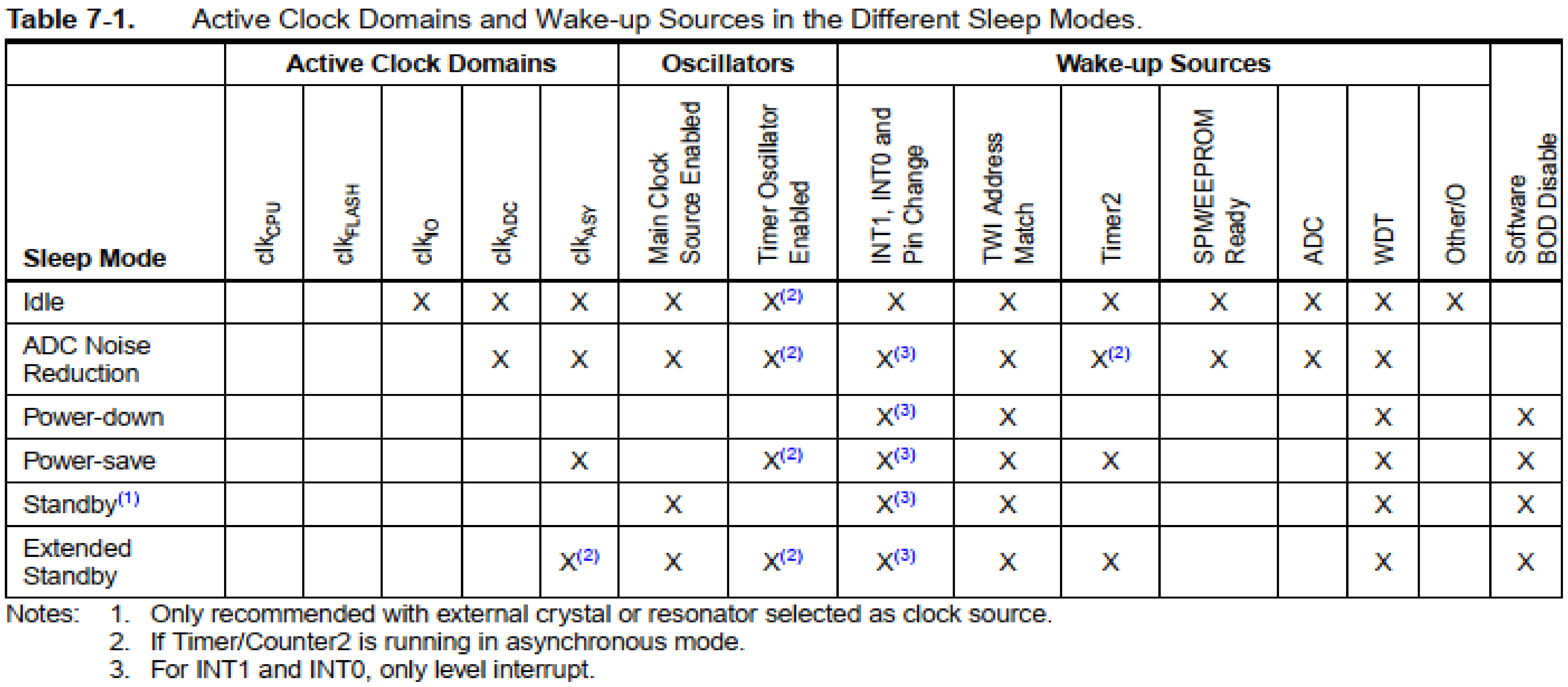


Table 7-1. Active Clock Domains and Wake-up Sources in the Different Sleep Modes.

	Active Clock Domains					Oscillators		Wake-up Sources							
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT1, INT0 and Pin Change	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other/O	Software BOD Disable
Idle			X	X	X	X	X[2]	X	X	X	X	X	X	X	
ADC Noise Reduction				X	X	X	X[2]	X[3]	X	X[2]	X	X	X		
Power-down								X[3]	X				X		X
Power-save					X		X[2]	X[3]	X	X			X		X
Standby[1]						X		X[3]	X				X		X
Extended Standby					X[2]	X	X[2]	X[3]	X	X			X		X

Notes: 1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.
3. For INT1 and INT0, only level interrupt.

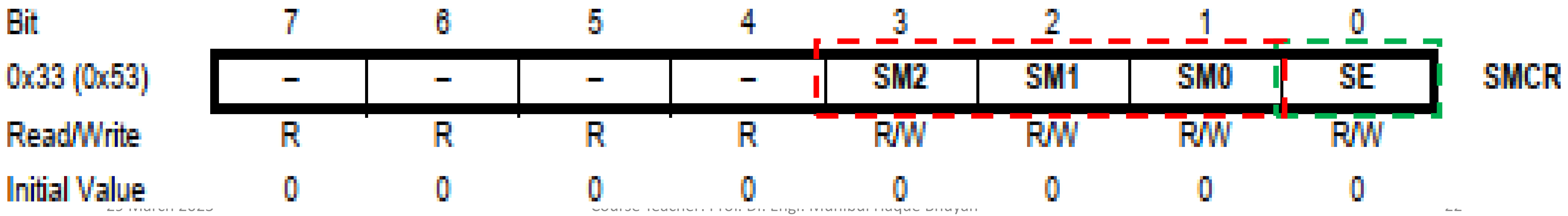
Register Description:

SMCR – Sleep Mode Control Register

The Sleep Mode Control Register (SMCR) contains control bits for power management. The function of these bits (SM2:0) is shown in the next slide.

For entering into any of the sleep modes, we need to enable the sleep bit in the Sleep Mode Control Register (SMCR.SE). Then, the Sleep Mode Select bits select any 1 of the sleep modes among 6 different sleep modes, viz. Idle, ADC Noise Reduction, Power-Down, Power-Save, Standby, and External Standby.

An internal or external Arduino interrupt or a Reset can wake up the Arduino from sleep mode.

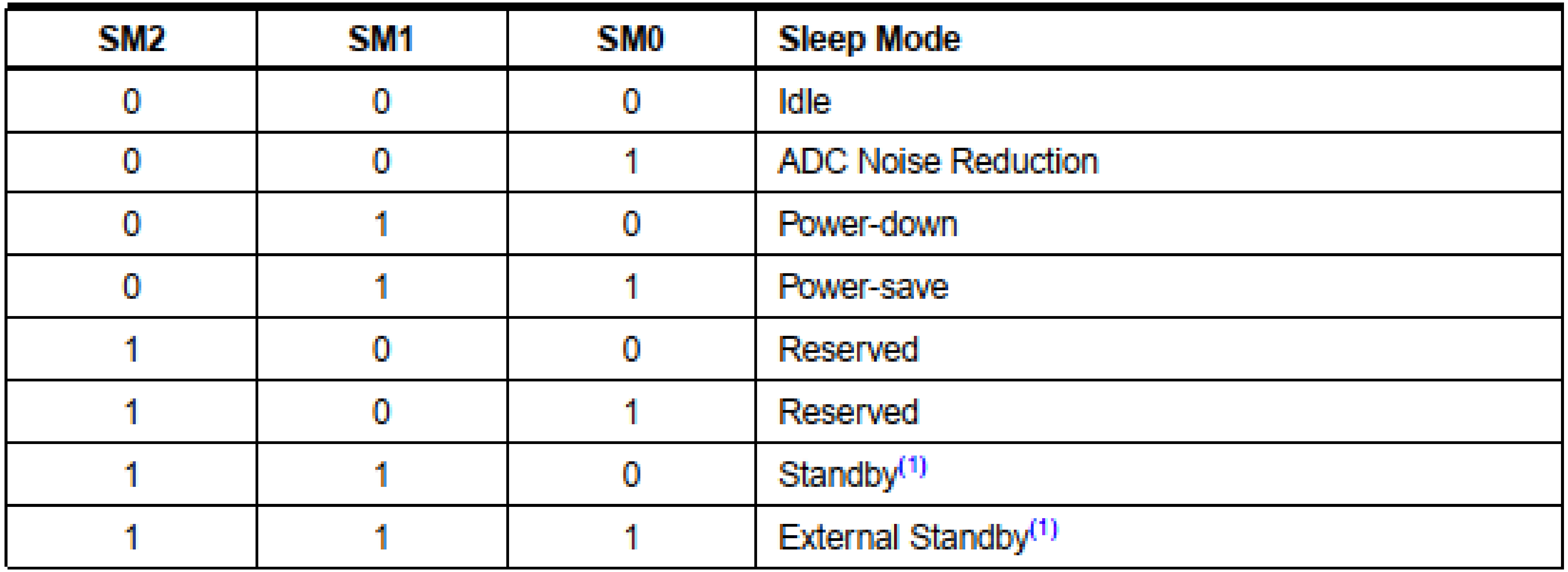


Bit	7	6	5	4	3	2	1	0	
0x33 (0x53)	–	–	–	–	SM2	SM1	SM0	SE	SMCR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Register Description:

```
0b00000111 =0x07
LDI R1, 0x07;
OUT SMCR, R1;
```

Table 7-2. Sleep Mode Select



SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby(1)
1	1	1	External Standby(1)

Note: 1. Standby mode is only recommended for use with external crystals or resonators.

Idle Mode:

For entering into the Idle sleep mode, write the SM[2,0] bits of the controller ‘000’. This mode stops the CPU but allows the SPI, 2-wire serial interface, USART, Watchdog, counters, and analog comparator to operate.
Idle mode basically stops the CLKCPU and CLKFLASH.
Arduino can be waked up at any time by using external or internal interrupts.

Arduino Code for Idle Sleep Mode:

```
LowPower.idle(SLEEP_8S, ADC_OFF, TIMER2_OFF, TIMER1_OFF,
TIMER0_OFF, SPI_OFF, USART0_OFF, TWI_OFF);
```

ADC Noise Reduction Mode:

To use this sleep mode, write the SM[2,0] bit to ‘001’. The mode stops the CPU but allows the ADC, external interrupt, USART, 2-wire serial interface, Watchdog, and counters to operate. ADC Noise Reduction mode basically stops the CLKCPU, CLKI/O, and CLKFLASH. We can wake up the controller from the ADC Noise Reduction mode by the following methods:

- External Reset
- Watchdog System Reset
- Watchdog Interrupt
- Brown-out Reset
- 2-wire Serial Interface address match
- External level interrupt on INT
- Pin change interrupt
- Timer/Counter interrupt
- SPM/EEPROM ready interrupt

Power-Down Mode:

Power-Down Mode stops all the generated clocks and allows only the operation of asynchronous modules. It can be enabled by writing the SM[2,0] bits to ‘010’. In this mode, the external oscillator turns OFF, but the 2-wire serial interface, watchdog, and external interrupt continue to operate. It can be disabled by only one of the methods below:

- External Reset
- Watchdog System Reset
- Watchdog Interrupt
- Brown-out Reset
- 2-wire Serial Interface address match
- External level interrupt on INT
- Pin change interrupt

Standard Low Power Example:

This is the simplest way of implementing the Low Power mode. It will use the LED as an indicator to tell if the device is in an active state or sleep state. The device will be in a sleep state for 5 seconds.

```
#include "ArduinoLowPower.h"

void setup() {
  pinMode(LED_BUILTIN, OUTPUT); }

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
  LowPower.sleep(5000);
}
```

Here we can also change a line of the code to perform the Deep Sleep mode of the device.

```
//LowPower.sleep(5000);
LowPower.deepSleep(5000);
```

Low Power-Power Down Mode Example:

In this sketch, the Arduino blinks an LED for 2 s and is then powered down for 2 s, and during that time the ADC and Brown-Out Detect (BOD) are disabled. When powered down, the Arduino's current drops from 14 mA, down to just 6 µA!

```
#include "LowPower.h"

void setup() {
  pinMode(13,OUTPUT);
}

void loop() {
  digitalWrite(13,HIGH);
  delay(2000);
  digitalWrite(13,LOW);
  LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);
}
```

Let’s load this sketch onto our Arduino, which is running with a 5 V supply at 16 MHz. To see how little current is needed in sleep mode, we are using an Arduino ATmega328P board to minimize the current.

Arduino Code for Power-Down Periodic Mode:

LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF);
The code is used to turn on the power-down mode. By using the above code, the Arduino will go to sleep for eight seconds and wake up automatically.
We can also use the power-down mode with an interrupt, where the Arduino will go to sleep but only wakes up when an external or internal interrupt is provided.

Arduino Code for Power-Down Interrupt Mode:

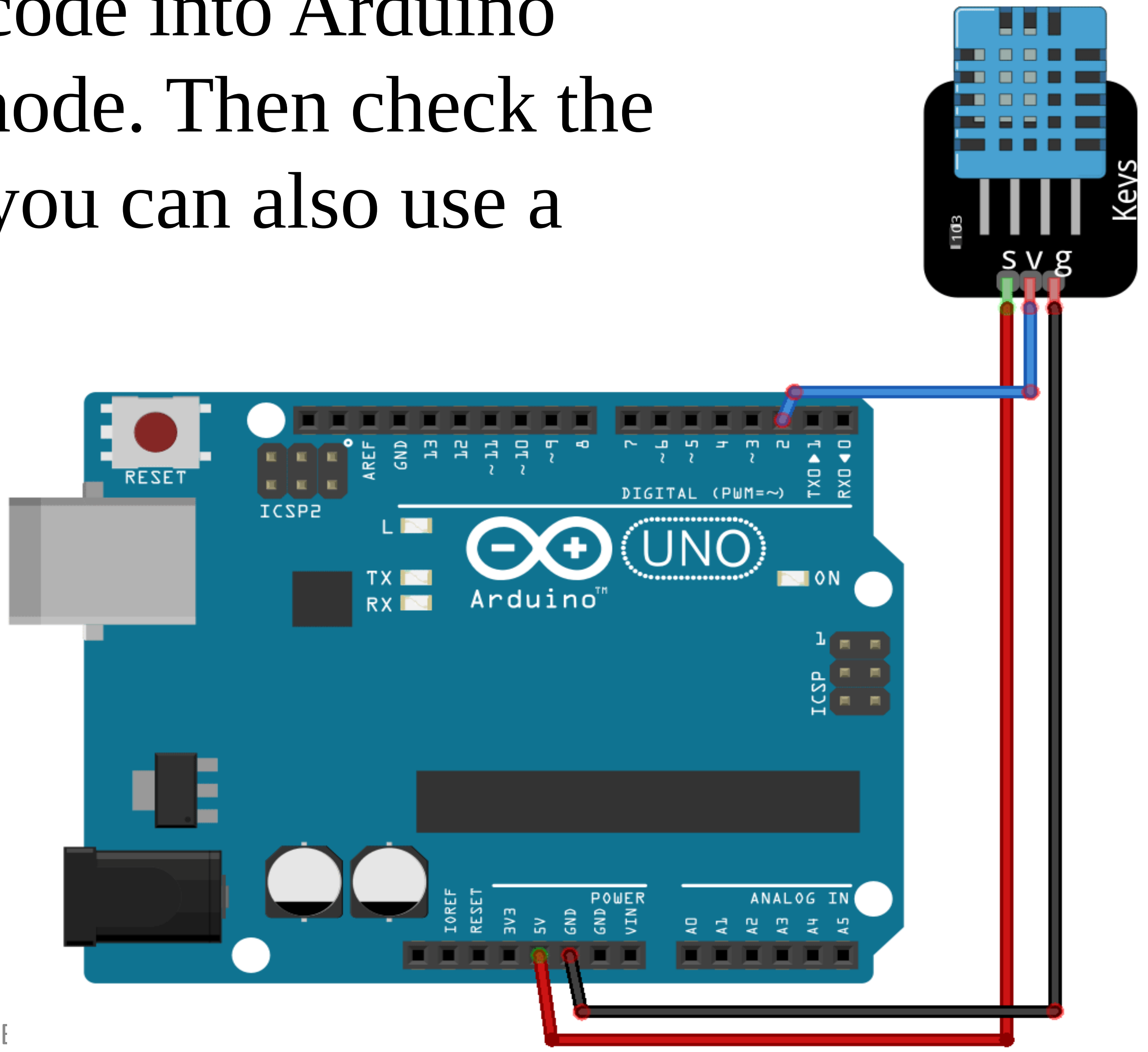
```
void loop() {
   // Allow the wake-up pin to trigger an interrupt on low.
   attachInterrupt(0, wakeUp, LOW);
   LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_OFF);
   // Disable external pin interrupt on the wake-up pin.
   detachInterrupt(0);
   // Do something here
}
```

Power-Save Mode: To enter the power-save mode, we need to write the SM[2,0] pin to ‘011’. This sleep mode is like the power-down mode, only with one exception i.e., if the timer/counter is enabled, it will remain in a running state even at the time of sleep. The device can be waked up by using the timer overflow bit, TOVn.
If you are not using the time/counter, it is recommended to use Power-down mode instead of power-save mode.
Standby Mode: The standby mode is identical to the Power-Down mode, the only difference in between them is the external oscillator kept running in this mode. For enabling this mode, write the SM[2,0] pin to ‘110’.
Extended Standby Mode: This mode is like the power-save mode only with one exception that the oscillator is kept running. The device will enter the Extended Standby mode when we write the SM[2,0] pin to ‘111’. The device will take six clock cycles to wake up from the extended standby mode.

Practical Application with a DHT11 Sensor:

Below are the requirements for this project, after connecting the circuit as per the circuit diagram. Upload the sleep mode code into Arduino using Arduino IDE. Arduino will enter idle sleep mode. Then check the current consumption into the USB ammeter. Else, you can also use a clamp meter for the same.

DHT11 Sensor



In this setup, to demonstrate Arduino Deep sleep modes, the Arduino is plugged into the **USB ammeter**. Then the USB ammeter is plugged into the USB port of the laptop.
The data pin of the DHT11 sensor is attached to the D2 pin of the Arduino.

Code Explanation

The code starts by including the library for the DHT11 sensor and the Low Power library. For downloading the Low Power library follow the link (*https://github.com/rocketscream/Low-Power*).
Then we defined the Arduino pin number to which the data pin of the DHT11 is connected and created a DHT object.

```
#include <dht.h>
#include <LowPower.h>

#define dataPin 2
dht DHT;
```

USB Ammeter

Code Explanation

In the void setup function, we have initiated the serial communication by using Serial.begin(9600), here the 9600 is the baud rate. We are using Arduino’s built-in LED as an indicator for the sleep mode. So, we have set the pin as OUTPUT and sent a LOW signal to this pin using the digitalWrite() command.

```
void setup() {
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);
}
```

Code Explanation

In the void loop function, we are making the built-in LED HIGH and the reading the temperature and humidity data from the sensor. Here, DHT.read11(); command is reading the data from the sensor. Once data is calculated, we can check the values by saving them into any variable. Here, we have taken two float type variables ‘t’ and ‘h’. Hence, the temperature and humidity data is printed serially on the serial monitor.

```
void loop() {
  Serial.println("Get Data From DHT11");
  delay(1000);
  digitalWrite(LED_BUILTIN, HIGH);
  int readData = DHT.read11(dataPin); // DHT11
```

Code Explanation

```
float t = DHT.temperature;
float h = DHT.humidity;

Serial.print("Temperature = ");
Serial.print(t);
Serial.print(" C | ");

Serial.print("Humidity = ");
Serial.print(h);
Serial.println(" % ");

delay(2000);
```

Code Explanation

Before enabling the sleep mode, we are printing "Arduino:- I am going for a Nap" and making the built-in LED LOW. After that, Arduino sleep mode is enabled by using the command mentioned below in the code.

The below code enables the idle periodic sleep mode of the Arduino and gives a sleep of eight seconds. It turns the ADC, Timers, SPI, USART, and 2-wire interface into the OFF condition.

Then, it automatically wakes up Arduino from sleep after 8 seconds and prints “Arduino:- Hey I just Woke up”.

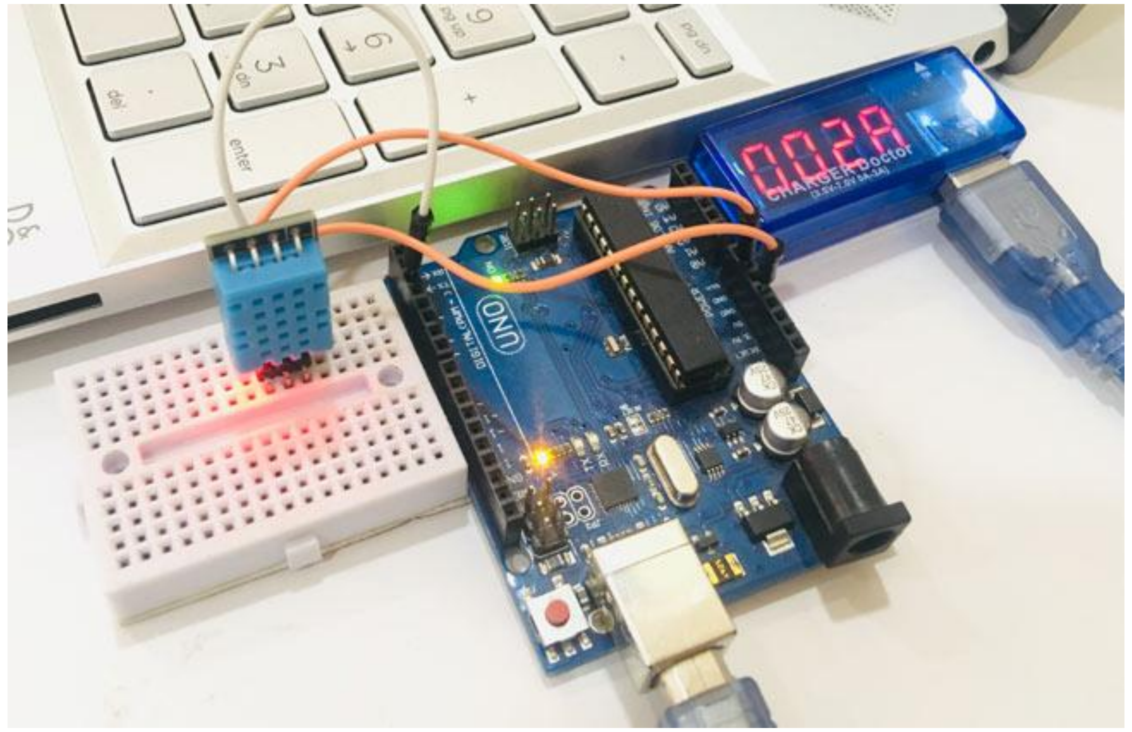
Code Explanation

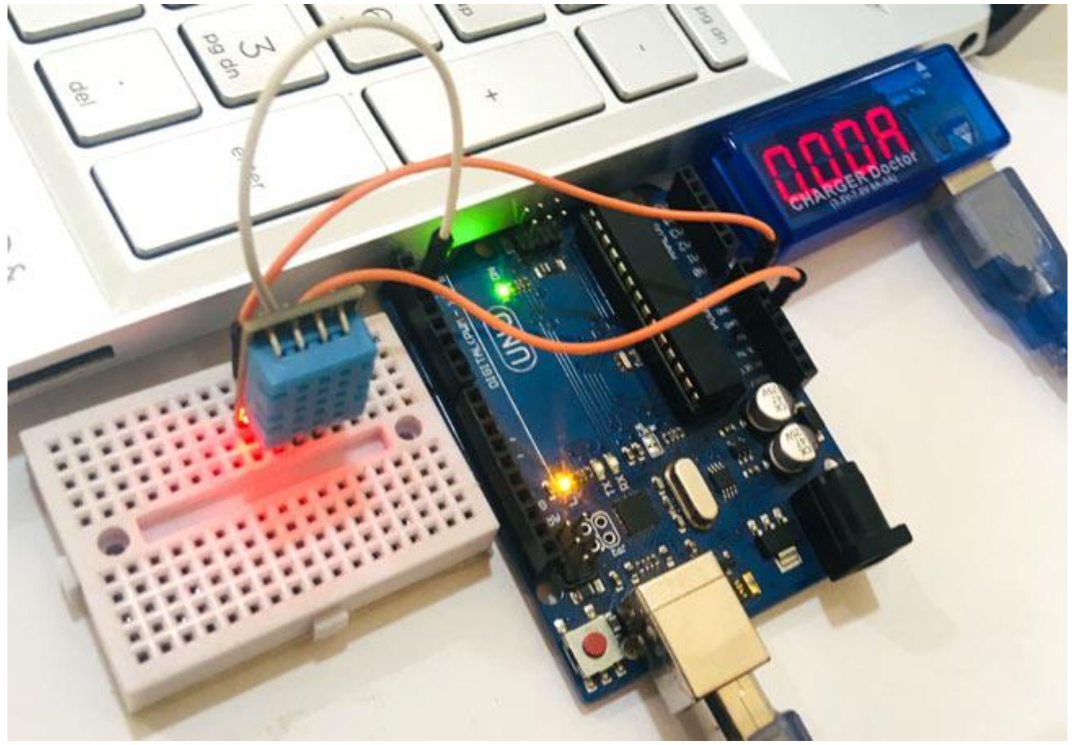
```
Serial.println("Arduino:- I am going for a Nap");
delay(1000);
digitalWrite(LED_BUILTIN,LOW);
LowPower.idle(SLEEP_8S, ADC_OFF, TIMER2_OFF, TIMER1_OFF, TIMER0_OFF,
SPI_OFF, USART0_OFF, TWI_OFF);
Serial.println("Arduino:- Hey I just Woke up");
Serial.println("");
delay(2000);
}
```

So, using this code, Arduino will only wake up for 24 (6×4 times of the loop) seconds in a minute and will remain in sleep mode for the rest of the 36 (= **60 – 24) seconds**, which significantly reduces the power consumed by the Arduino weather station. Therefore, using the Arduino with the sleep mode can approximately double the device runtime.

Experimental Setup with USB Ammeter

USB ammeter is a plug-and-play device used to measure the voltage and current from any USB port. The dongle plugs in between the USB power supply (computer USB port) and the USB device (Arduino). This device has a 0.05 Ω resistor in-line with the power pin through which it measures the value of the current drawn. The device comes with four seven-segment displays, which instantly display the values of current and voltage consumed by the attached device. These values flip in an interval of every 3 seconds.





Example 1

Compute the total time for which the device is in low-power mode as per the following program if the loop continues for 20 cycles.

```
#include "ArduinoLowPower.h"

void setup() {
  pinMode(LED_BUILTIN, OUTPUT); }

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
  LowPower.deepSleep(8000);  }
```

Answer:
If the above program is run then the time for which the device is in low-power deep sleep mode is 8000 ms or 8 s. Hence, if the loop continues for 20 cycles, then the device is in low-power deep sleep mode for 8×20 = 160 s.

Example 2

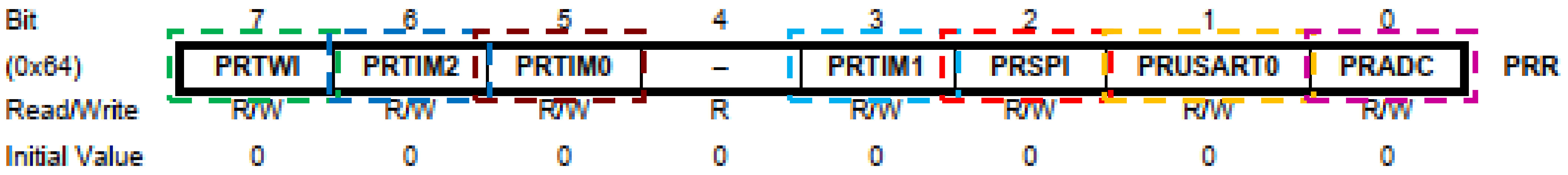
Compute the duration that the Arduino blinks an LED connected to pin 10 and makes the power down the system. When powered down, the Arduino’s current drops from 10 mA down to just 8 mA. If the Arduino’s supply voltage is 3.3 V then compute the amount of power that is saved by the Arduino during this power down mode.

```
#include "LowPower.h"

void setup() {
  pinMode(10, OUTPUT); }

void loop() {
  digitalWrite(10, HIGH);
  delay(1000);
  digitalWrite(10, LOW);
  LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF, SPI_OFF, TWI_OFF);
}
```

Answer:
If the above program is run then the time for which the LED is ON is 1000 ms or 1 s because the digitalWrite() function sends a ‘HIGH’ signal to pin 10 where an LED is connected and then the delay() function makes a delay of 1000 ms. After that, the digitalWrite() function sends a ‘LOW’ signal to pin 10. After that, there is no delay() function, but the device goes into power-down mode for 4 s. As such, the duration that the Arduino blinks an LED connected to pin 10 is every 5 s, that is, the LED is ON for 1 s and OFF for 4 s, and this loop continues.

Initially, when the power is not down then the device draws a 10 mA current. Hence, it consumes a power of 3.3×10 = 33 mW.
When the device is in low-power power-down mode then the device draws an 8 mA current. Hence, it consumes a power of 3.3×8 = 26.4 mW = 0.0264 mW.
So, the amount of power that is saved by the Arduino during the low-power power-down mode = 33 – 0.0264 = 32.9736 mW.

PRR: Power Reduction Register:



Bit	7	6	5	4	3	2	1	0	
(0x64)	PRTWI	PRTIM2	PRTIM0	–	PRTIM1	PRSPI	PRUSART0	PRADC	PRR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 - PRTWI: Power Reduction TWI
- Bit 6 - PRTIM2: Power Reduction Timer/Counter2
- Bit 5 - PRTIM0: Power Reduction Timer/Counter0
- Bit 4 - Res: Reserved bit
- Bit 3 - PRTIM1: Power Reduction Timer/Counter1
- Bit 2 - PRSPI: Power Reduction Serial Peripheral Interface
- Bit 1 - PRUSART0: Power Reduction USART0
- Bit 0 - PRADC: Power Reduction ADC

```
0b10101100 =0xAC
LDI R10, 0xAC;
OUT PRR, R10;
IN R10, PRR;
```

Thanks for Attending....

[(6) Reducing Arduino's Power Consumption Part 1 – YouTube](https://www.youtube.com/watch?v=ktvUunBQQD0)
https://www.youtube.com/watch?v=ktvUunBQQD0