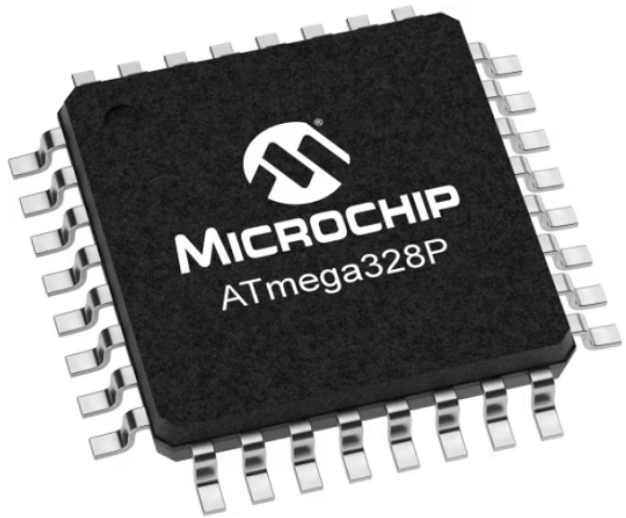
Beyond the Libraries: Developing Pure C Drivers from Scratch for ATmega328P

Author: Hamza Taous



Outline

- Object of the Documentation
- Why Develop Drivers from Scratch?
- Brief Reminder on the C Compilation Process
- Overview of ATmega328P MCU
- Required Tools and Software
- Setting Up the Environment
- GPIO Driver Development
- ADC Driver Development
- Timers Driver Development
- UART Driver Development

Object of the Documentation

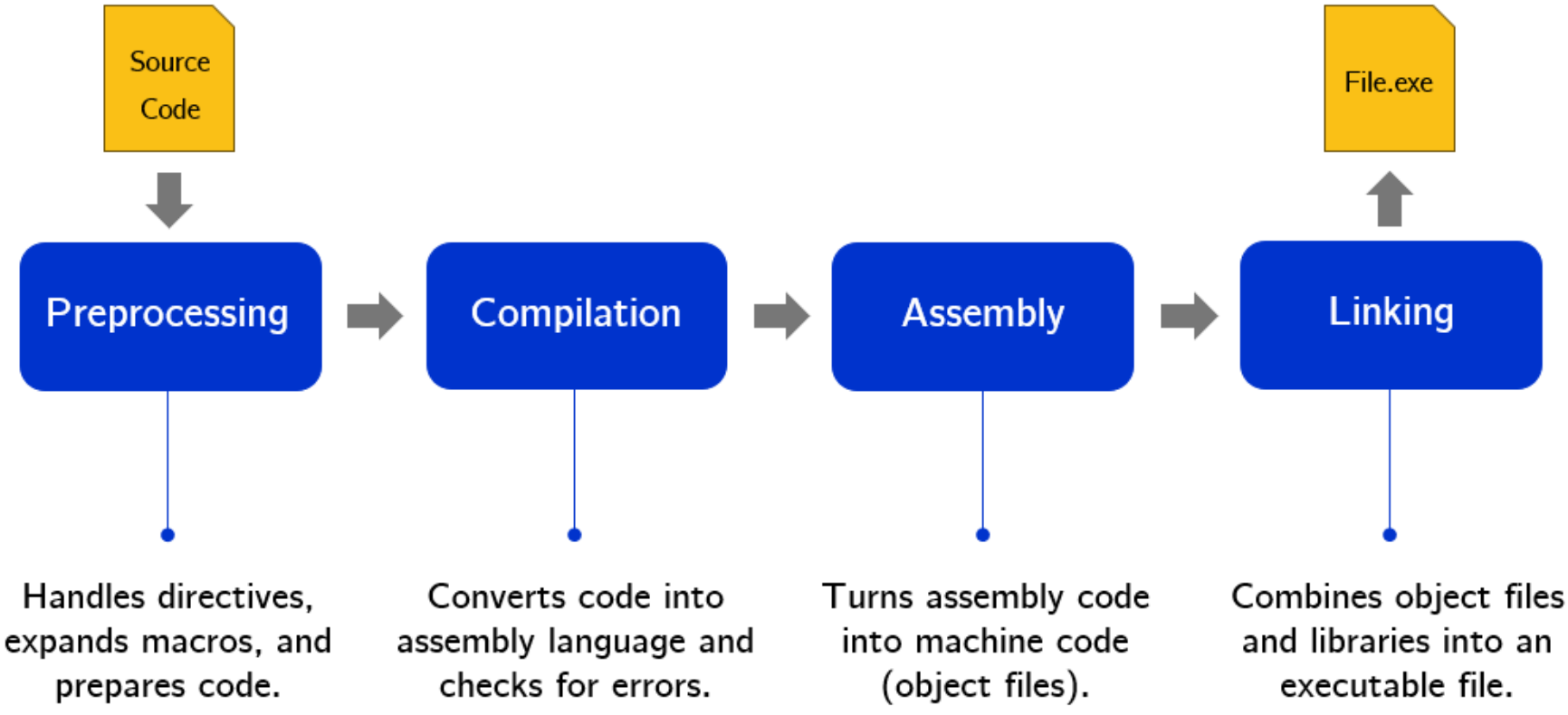
- This documentation serves as a practical guide for embedded system developers and enthusiasts, aiming to provide a detailed, hands-on approach to developing custom C drivers for the ATmega328P microcontroller without relying on any third-party libraries or predefined functions.
- This guide will lead you through the process of configuring and utilizing the microcontroller's peripherals directly, using only the ATmega328P's registers and datasheets.
- By the end of this documentation, you will have written custom drivers for critical peripherals, understand how to configure and manipulate peripheral registers to achieve desired functionality and gain the confidence to apply similar techniques to other microcontrollers and embedded systems, enabling you to create fully customized solutions for your hardware.

Why Develop Drivers from Scratch?

- **Performance Optimization**
  - Efficient use of MCU's resources
  - Reduced code and faster execution
- **Greater Flexibility and Control**
  - More control over the hardware's behavior
  - Seamless adaptation to changes
- **Independence from Third Party Code**
  - No black box code
  - Improved security
- **Enhanced Learning and Understanding**
  - In-depth knowledge of MCU operations
  - Development of analytical and problem-solving skills

Brief Reminder on the C Compilation Process

- The C compilation process is a sequence of steps that converts C source code into an executable program.



Overview of ATmega328P MCU

- The ATmega328P is a popular microcontroller from Microchip's AVR family, widely known for its use in hobbyist electronics, especially in Arduino boards like the Arduino Uno.

- As main features, it includes an 8-bit AVR RISC processor, 32 KB of flash memory, 23 I/O pins, 3 timers, a watchdog timer, a 10-bit ADC, integrated communication interfaces (UART, SPI, and I^2C), all within a power efficient architecture.

- It's ideal for those looking to start in embedded systems due to its simplicity, rich set of peripherals, and strong ecosystem. It can be programmed using low level C or assembly, giving users full control over its operation, which is particularly appealing when developing drivers from scratch.

- For more details about this MCU, refer to its automotive-grade datasheet, which will be our primary reference throughout this documentation.

Required Tools and Software

VS Code

A versatile code editor that provides features like syntax highlighting, debugging support, and extensions to enhance productivity while writing and managing your C code.

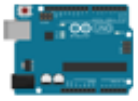
AVR-GCC

A collection of tools that includes a compiler for converting C code into machine code that the ATmega328P can execute, along with additional tools for linking and assembling the code.

AVRDUDE

A CLU for uploading the compiled code to a physical ATmega328P microcontroller, such as the one on an Arduino Uno board, in case you want to flash and test the code on real hardware.

Arduino Uno



The most popular board that features the ATmega328P microcontroller. It can be used to run tests on real hardware, though this is entirely optional based on your preference.

Required Tools and Software [Additional Insights]

- There are several tools and software that can be used to achieve the same results when developing drivers. Some offer all-in-one solutions that allow us to write, compile, and simulate our code in a single environment (e.g., Proteus ISIS, Tinkercad Circuits, Wokwi).

- Since the objective of this documentation is to build drivers for the ATmega328P from the ground up, using command line tools like AVR-GCC toolchain and AVRDUDE aligns perfectly with the need to understand the process deeply and take full control over it.

- While single button tools can be convenient, such tools are more advantageous for low level development and helps you become familiar with some command lines, enhancing your skills as an embedded systems engineer.

"Smooth seas don't make good sailors." — *African Proverb*

Setting Up the Environment

1. **Installing VS Code :**

   Visit the official VS Code website, download the installer for your OS and follow the on screen instructions to install the software.

2. **Installing the AVR-GCC Toolchain :**

   Download AVR-GCC from the official AVR toolchain page, follow the provided installation guide and once installed, ensure that the AVR-GCC is available in your terminal or command prompt. To verify, you can make a quick check on the version by running the following command :

   ```
   avr-gcc --version
   ```

3. **Installing AVRDUDE :**

   Get the installer for Windows from this link, then run it and follow the steps to install. A portable version is also available (usable on any OS).

GPIO Driver Development

- In this section, we will develop our 1st peripheral driver for GPIO (General Purpose Input/Output) on the ATmega328P.

GPIO Driver Development

- GPIO enables the microcontroller to interact with external devices by setting pins as inputs or outputs, allowing it to read signals or control components like LEDs, switches, and more.

- On the ATmega328P, GPIO is handled by three main ports: **Port B** (PB0–PB7), **Port C** (PC0–PC6), and **Port D** (PD0–PD7). Each GPIO pin has three registers associated with it: **DDRx** (Data Direction Register), **PORTx** (Port Register) and **PINx** (Pin Input Register).

  → **DDRx:** Configures the pin as input (0) or output (1).
  → **PORTx:** Sets the output level or activates pull-up resistors.
  → **PINx:** Reads the input value of the pin.

- These registers are part of the I/O memory space in the SRAM, located at addresses from 0x20 to 0x5F (Datasheet - Section 7.2 on Page 18).

GPIO Driver Development

Below is an illustration that shows the GPIO registers along with their memory addresses (Datasheet - Section 13.4 on Pages 72 & 73). **DDRx** and **PORTx** are initialized to 0x00 and are both readable and writable, while **PINx** are read only, showing the actual input state of the pins.

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
0x08 (0x28)	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	PORTC
0x07 (0x27)	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	DDRC
0x06 (0x26)	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	PINC
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
0x09 (0x29)	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND

GPIO Driver Development

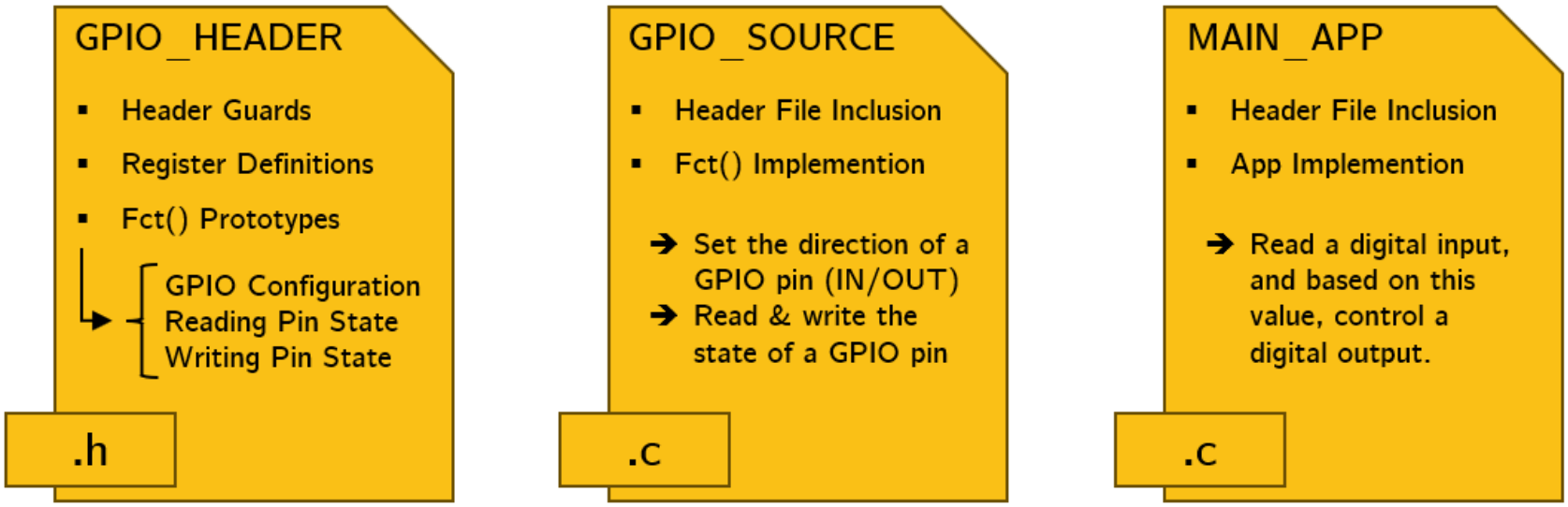
Now, let's outline our strategy for building the driver!

- To improve organization and maintainability, three files will be created: a **header file (.h)** for function declarations, a **source file (.c)** for implementation, and a **main application file (.c)** for for application logic. This approach will be used for the upcoming drivers we develop as well.

- Exceptionally for this driver and to deepen our understanding while increasing the development challenge, we will avoid using the <**avr/io.h**> header, which is part of the AVR-GCC toolchain and serves as a bridge between C code and the AVR microcontroller, making registers more human readable by providing their names. Instead, we will directly access registers through their memory addresses, but for the next drivers, we will return to using <**avr/io.h**> for easier register handling.

GPIO Driver Development

- As a best practice in embedded systems, we will use the keyword **volatile** for register access to prevent compiler optimizations and ensure we access the actual value of the registers.
- The illustration below highlights the contents of each of the three files :



GPIO Driver Development

Time to run the driver test!

1. Open AVR-GCC in the directory containing the source files and generate the **.hex** file using the following two commands :

```
avr-gcc -g -Os -DF_CPU=16000000UL -mmcu=atmega328p
-o APP_GPIO.elf APP_GPIO.c GPIO.c; avr-objcopy -O
ihex -R .eeprom APP_GPIO.elf APP_GPIO.hex
```

2. For ease, upload the **.hex** file to a simulated ATmega328P and begin testing. A schematic in ISIS Proteus is already prepared, where the application reads the input voltage on PORTB6 to control an LED on PORTB1.

§ Find the driver files and ISIS schematic in the GitHub link below : https://tinyurl.com/mt8ate9e

ADC Driver Development

- In this section, we will detail the process of developing an ADC driver from scratch for the ATmega328P, from configuration to testing.

ADC Driver Development

- The ADC (Analog to Digital Converter) is essential in microcontrollers, converting analog signals like temperature, light, or sound into digital values, allowing the microcontroller to interface with sensors and perform control tasks based on those measurements.
- The ADC on the ATmega328P has eight input channels and a fixed resolution of 10 bits. It is controlled by the following registers :

  → **ADMUX (Analog Multiplexer Selection Register):** Configures the reference voltage and selects the input channel.
  → **ADCSRA (ADC Control and Status Register A):** Enables the ADC, controls the conversion process, and sets the ADC prescaler.
  → **ADCL (Data Register Low) & ADCH (Data Register High):** Store the 10-bit ADC result in ADCL (low) and ADCH (high).
  → **ADCSRB (ADC Control and Status Register B):** Configures the ADC for different operating modes.

ADC Driver Development

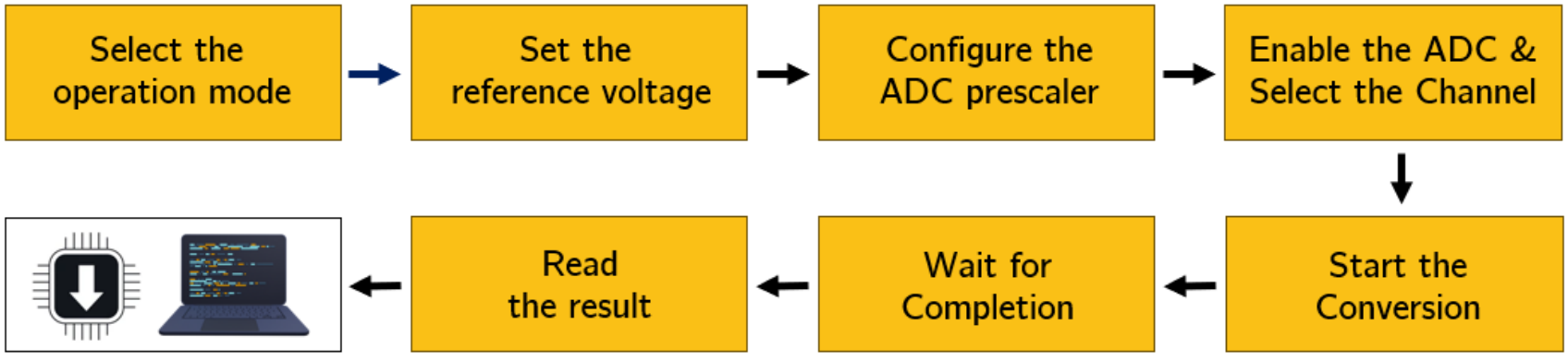
- These registers are in the extended I/O memory space of SRAM, located at addresses from 0x60 to 0xFF (Datasheet - Section 7.2 on Page 18).
- All these registers are readable and writable. In contrast, while **ADCL** and **ADCH** primarily store the ADC conversion result, they can also be written to, enabling their values to be reset and helping to prevent the processing of outdated data. Typically, these registers are read after a conversion to access the ADC result.
- The following picture shows the ADC registers with their memory addresses (Datasheet - Section 23.9 on Pages 217, 218, 219 & 220).

(0x7C)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0
(0x7B)	ADCSRB	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0
(0x7A)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
(0x79)	ADCH	ADC data register high byte							
(0x78)	ADCL	ADC data register low byte							

ADC Driver Development

Time to tackle the strategy for coding success!

- After conducting a comprehensive review of the ADC configuration registers and the pertinent information detailed in the datasheet, these are the key steps to follow for setting up the ADC :



- First, we will adopt single conversion mode, using only the **ADCSRA** register to manually initiate each ADC conversion through code. In contrast to automatic modes, this configuration is ideal in situations that demand

ADC Driver Development

high accuracy or specific conditions to be met before taking a reading, as it reduces the complexities and unpredictability associated with automatic triggering mechanisms in the **ADCSRB** register.

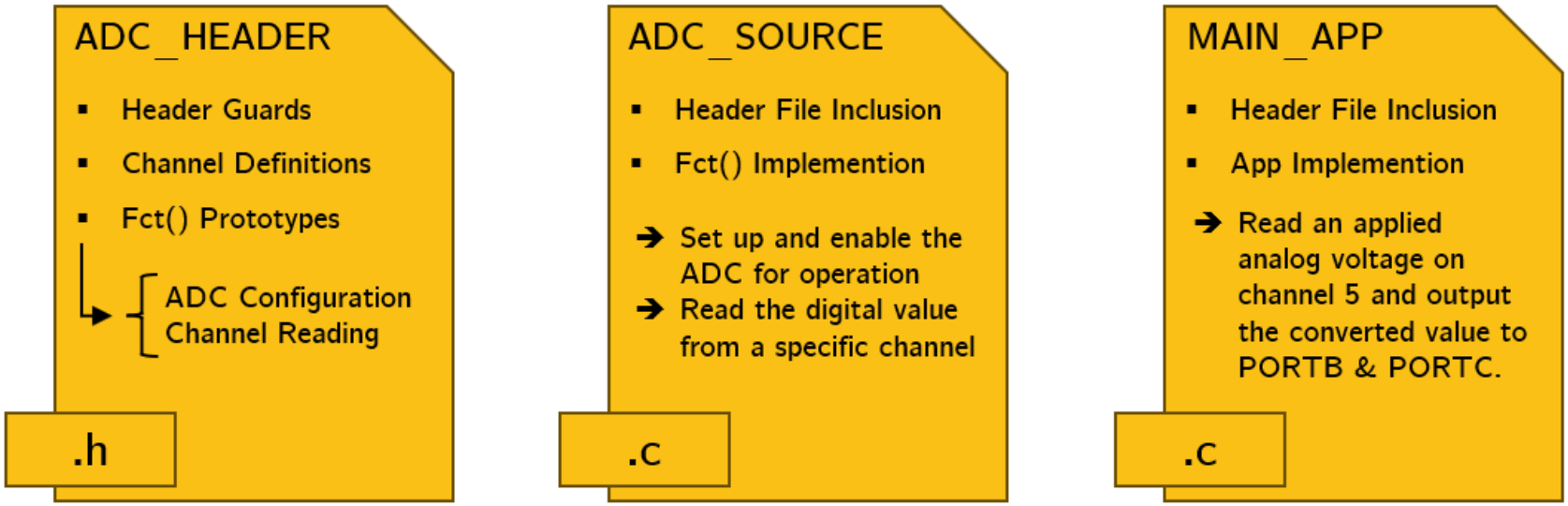
- As the reference voltage for the ADC, we will opt for **AVcc** due to its simplicity, stability, the ability to handle a wider range of measurements at 5V, and the elimination of the need for additional components.

- Despite our focus on ADC, it's important to note that the ATmega328P's ADC pins can trigger interrupts (Datasheet - Section 1 on Page 3), but only if configured and not automatically. In any case, we will clear the I-bit in the **SREG** register to disable the overall interrupt system and ensure accurate and reliable ADC readings.

- For a balance between sampling speed and conversion accuracy, we will use a prescaler of 128 at 16 MHz or 64 at 8 MHz.

ADC Driver Development

- In the ATmega328P, the DIP version (like on Arduino Uno) has only 6 accessible ADC channels, while the TQFP package offers 8. Our driver will support all 8 channels to ensure compatibility with both packages.
- The illustration below showcases the structure of the ADC driver files :



ADC Driver Development

Let's see our ADC driver in action!

1. As before, run the same commands mentioned in slide 15, making the necessary modifications, to generate the **.hex** file for testing.

2. Upload the **.hex** file to a simulated ATmega328P to ease the testing process. A prepared ISIS schematic available on GitHub to test the driver.

3. Initiate the simulation and rotate the potentiometer to observe how the changes in its position are reflected on PORTB and the first two pins of PORTC, clearly demonstrating the ADC's response to the varying input voltage and its effect on the corresponding digital output.

§ Driver files and ISIS schematic are in the GitHub link below : https://tinyurl.com/mr2pc6rf

Timers Driver Development

- In this part, we will build a timer driver from scratch for the ATmega328P, with a focus on configuration, implementation, and testing.

Timers Driver Development

- Timers are core components in microcontrollers that enable time measurement, event scheduling, PWM signal generation, delay management, and interrupt handling, all vital for effective control in embedded systems.
- The ATmega328P has three timers :
  - → **Timer0:** An 8 bit timer, primarily used for basic timing functions.
  - → **Timer1:** A 16 bit timer, suitable for more precise timing applications.
  - → **Timer2:** Another 8 bit timer with some additional features.
- Each timer is managed by its own registers :
  - → **TCCRnA/B** registers to configure timer's mode, clock, and output.
  - → **TCNTn** registers to store the current count value of the timer.
  - → **OCRnA/B** registers to define values for PWM or interrupt actions.
  - → **TIMSKn** registers to enable interrupts for timer related events.
  - → **TIFRn** registers to hold flags for timer events.

Timers Driver Development

- **Timer1** stands out from **Timer0** and **Timer2** mainly because to its 16 bit architecture, which provides a wider range of registers for more advanced control, making it more flexible for tasks requiring accurate timing, high resolution PWM generation, and advanced modes like Input Capture, which are unavailable in the 8 bit timers.
- According to the datasheet (Section 14, 15 & 17), below is an illustration showing the timer's behavior across different modes :

Normal Mode	Counts up to the max value, overflows, and restarts from zero.
CTC Match Mode	Counts up to a specified value in the OCR register, resets to zero on match, and triggers an interrupt.
PWM Mode	Counts up (or down) to a value and produces a PWM with a variable duty cycle controlled by OCR.
Input Capture Mode (T1 only)	Captures the current timer count value when an external event occurs on the input capture pin.

Timers Driver Development

- In this portion, we will focus exclusively on **Timer1** in our driver development, due to its precision and advanced capabilities.
- The **Timer1** registers are located in the I/O memory range of SRAM (Datasheet - Section 7.2 & 30 on Pages 18 & 278). Below is a picture showing these registers with their corresponding addresses :

(0x8B)	OCR1BH	Timer/Counter1 - Output compare register B high byte							
(0x8A)	OCR1BL	Timer/Counter1 - Output compare register B low byte							
(0x89)	OCR1AH	Timer/Counter1 - Output compare register A high byte							
(0x88)	OCR1AL	Timer/Counter1 - Output compare register A low byte							
(0x87)	ICR1H	Timer/Counter1 - Input capture register high byte							
(0x86)	ICR1L	Timer/Counter1 - Input capture register low byte							
(0x85)	TCNT1H	Timer/Counter1 - Counter register high byte							
(0x84)	TCNT1L	Timer/Counter1 - Counter register low byte							
(0x82)	TCCR1C	FOC1A	FOC1B	–	–	–	–	–	–
(0x81)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
(0x80)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10
(0x6F)	TIMSK1	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1
0x16 (0x36)	TIFR1	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1

Timers Driver Development

Let's explore our approach to unlock Timer1's full potential!

- In the first place, the driver will encompass the key features of the timer : delay management, event counting, and PWM signal generation. Based on the datasheet, the following settings need to be configured for each mode, as shown in the representation below :

Timer Mode	Counter Mode	PWM Mode
• Choose the mode • Select the prescaler • Set the initial value • Calculate the number of ticks required for the delay	• Set the mode • Configure the timer to use an external clock on the T1 pin • Select the edge trigger • Put the target count • Define the start value	• Configure the output pins (OC1A & OC1B) • Select the PWM mode and handle its settings • Define the duty cycle • Choose the prescaler • Configure the output compare behavior

Timers Driver Development

- Instead of handling the **Timer1** registers one by one, we will use the combined register feature in <**avr/io.h**>.
- We will use CTC mode to handle delays because it enables the timer to reset at a fixed compare match value, unlike normal mode, which depends on overflows. This approach reduces cumulative timing errors and increases precision, especially for longer delays.
- While a prescaler of 64 provides an optimal balance of range and accuracy for managing delays, the driver will support any prescaler, configurable through an argument in the delay function call.
- In counter mode, we don't have the luxury of choosing the input pin due to the hardware design of the ATmega328P. **Timer1** requires PORTD5, and **Timer0** (if used) requires PORTD4, as each timer accepts input only from its designated pin. However, the driver allows selecting the edge type (rising or falling) for event counting.

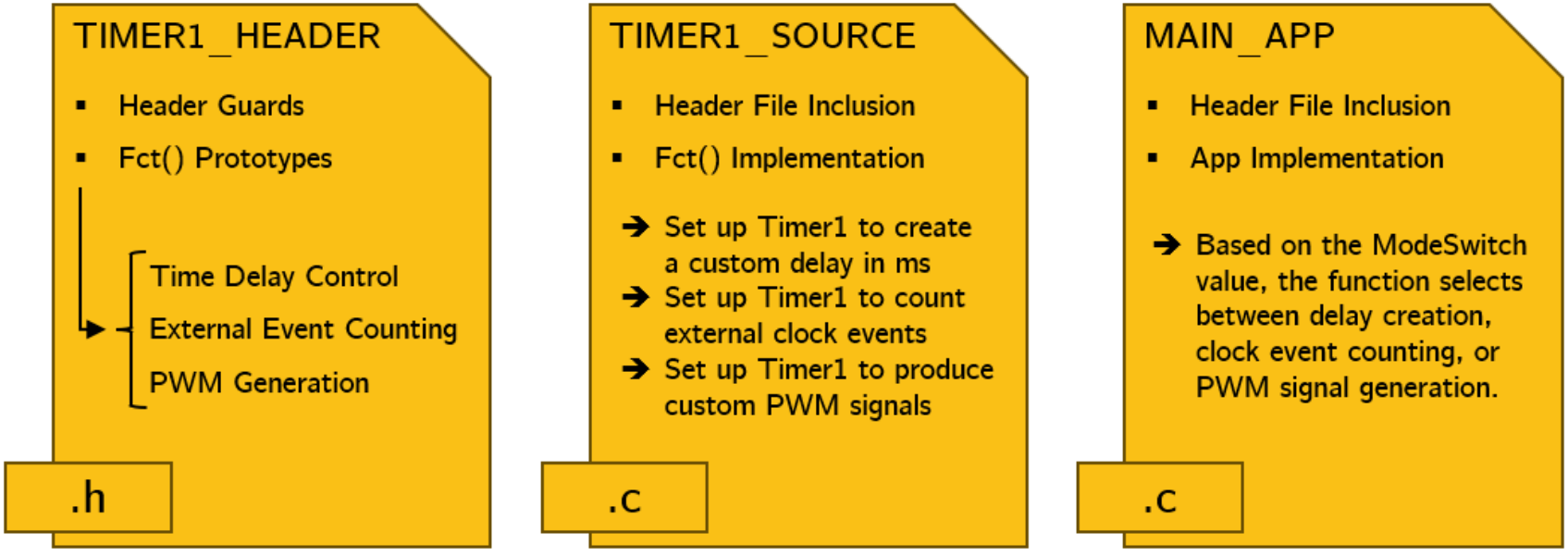
Timers Driver Development

- For PWM signal generation, we will implement Fast PWM with ICR1 as TOP, ensuring 16 bit resolution for accurate duty cycle control, frequency adjustments, and enhanced synchronization with external components.
- As in counter mode, we are limited to using only PORTB1 and PORTB2 for PWM generation in **Timer1** due to hardware constraints. Meanwhile, the driver allows for the adjustment of the duty cycle, prescaler, PWM frequency, pin selection (one or two), compare output mode, and signal toggling, all through function arguments.
- The driver will handle the ICR1 top value calculation, but we should be mindful when calling the function to avoid selecting a PWM frequency and prescaler that would result in an ICR1 value greater than 65535. The formula below can help with this :

$$\mathbf{ICR1} = \frac{\mathbf{F}_{\mathrm{CPU}}}{\mathbf{prescaler} \times \mathbf{frequency}} - 1$$

Timers Driver Development

- Each function in our driver is programmed to validate and handle invalid input values. For example, in PWM mode, if the prescaler and PWM frequency result in an ICR1 value that exceeds its limit, an automatic adjustment will be made to bring it within the valid range.
- The illustration below highlights the main elements within the driver files :



Timers Driver Development

Time to witness Timer1 perform!

1. Pick the **ModeSwitch** variable and function settings.
2. Generate the **.hex** file (check slide 15 for commands).
3. As usual, a prepared ISIS schematic is available for ease of testing. Upload the **.hex** file and start testing based on the **ModeSwitch** variable value :
   - ✓ **ModeSwitch = 1:** Try different delays and ensure their accuracy.
   - ✓ **ModeSwitch = 2:** Use multiple counts for both edge types and make sure the LED on PORTB5 toggles with each target count reached.
   - ✓ **ModeSwitch = 3:** Test PWM with a single pin, different duty cycles and frequencies, both output modes, and the toggling feature.

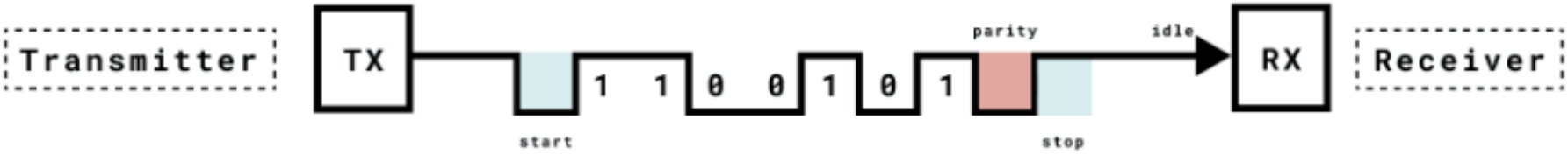
§ Check the GitHub link below for the driver files and ISIS schematic : https://tinyurl.com/4r8vxy4j

UART Driver Development

- In this part, we will craft a UART driver from scratch for the ATmega328P, covering configuration, implementation, and testing.

UART Driver Development

- UART is a hardware protocol for asynchronous serial communication between devices, using two wires (Tx & Rx) to exchange data. It operates without a shared clock, relying on the timing of the sender and receiver.
- In UART, data is sent in packets with a start bit, data bits, optional parity bit, and stop bits. The baud rate (data transmission speed) is a critical factor in ensuring accurate communication between the devices.



- The ATmega328P actually has a USART peripheral, which supports both UART (asynchronous) and USART (synchronous) modes. However, we will focus on developing the driver using UART, as it is the mode used by the most commonly used predefined functions in Arduino.

UART Driver Development

- Just to note, the ATmega328P's USART can be configured to SPI mode, allowing it to function as an SPI interface and enabling high speed, synchronous communication with SPI compatible devices. This feature is particularly useful in scenarios where no dedicated SPI peripheral is available or when the available SPI modules are already in full active use (Datasheet - Section 20 on Page 166).
- The ATmega328P includes a single USART peripheral. Its operation is managed by the following key registers :
  - → **UCSRnA (USART Control and Status Register A):** Monitors USART status, including buffer states and transmission completion.
  - → **UCSRnB (USART Control and Status Register B):** Controls transmitter, receiver, interrupts, and character size.
  - → **UCSRnC (USART Control and Status Register C):** Configures data frame format, parity, stop bits, and mode.

UART Driver Development

→ **UDRn (USART Data Register):** Stores data required for transmission or reception during serial communication.

→ **UBBRn (USART Baud Rate Register):** Sets the baud rate.

- These registers reside in the I/O memory space within the ATmega328P's SRAM (Datasheet - Section 13.4 on Pages 72 & 73), and are initialized to 0x00 after a system reset. Most of the register bits are read-write, while certain status bits are read only. Below is an illustration of the USART registers along with their respective memory addresses :

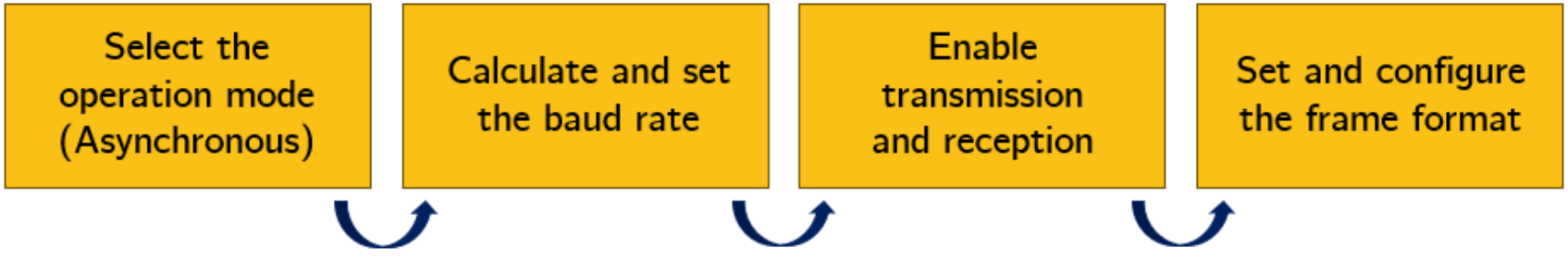
(0xC6)	UDR0	USART I/O data register							
(0xC5)	UBRR0H					USART baud rate register high			
(0xC4)	UBRR0L	USART baud rate register low							
(0xC3)	Reserved	–	–	–	–	–	–	–	–
(0xC2)	UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 /UDORD0	UCSZ00 / UCPHA0	UCPOL0
(0xC1)	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
(0xC0)	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0

UART Driver Development

Time to build up a robust UART communication driver!

- First, our driver will be built around functions for UART initialization, checking data availability, reading bytes, transmitting data, and sending strings in order to transmit a complete message with **Timer1** used for delays to prevent overwhelming the serial communication.

- Considering the reasons highlighted earlier, as well as its simplicity, asynchronous mode is our choice for this setup. The following illustration depicts the configuration steps necessary to establish communication with other devices using UART :



UART Driver Development

- The mode selection bits are cleared upon reset, meaning no configuration is required since the ATmega328P defaults to asynchronous mode (Datasheet - Section 19.10.4, Table 19.4 on Page 161).

- For setting the baud rate, common choices like 9600 bps and 115200 bps are preferred because they strike a balance between speed, reliability, and compatibility across devices, with minimal transmission errors. These values reflect the desired speed, but the actual communication speed is determined by the microcontroller using the following **UBRR** register formula (Datasheet - Section 19.13.1, Table 19.1 on Page 146) :

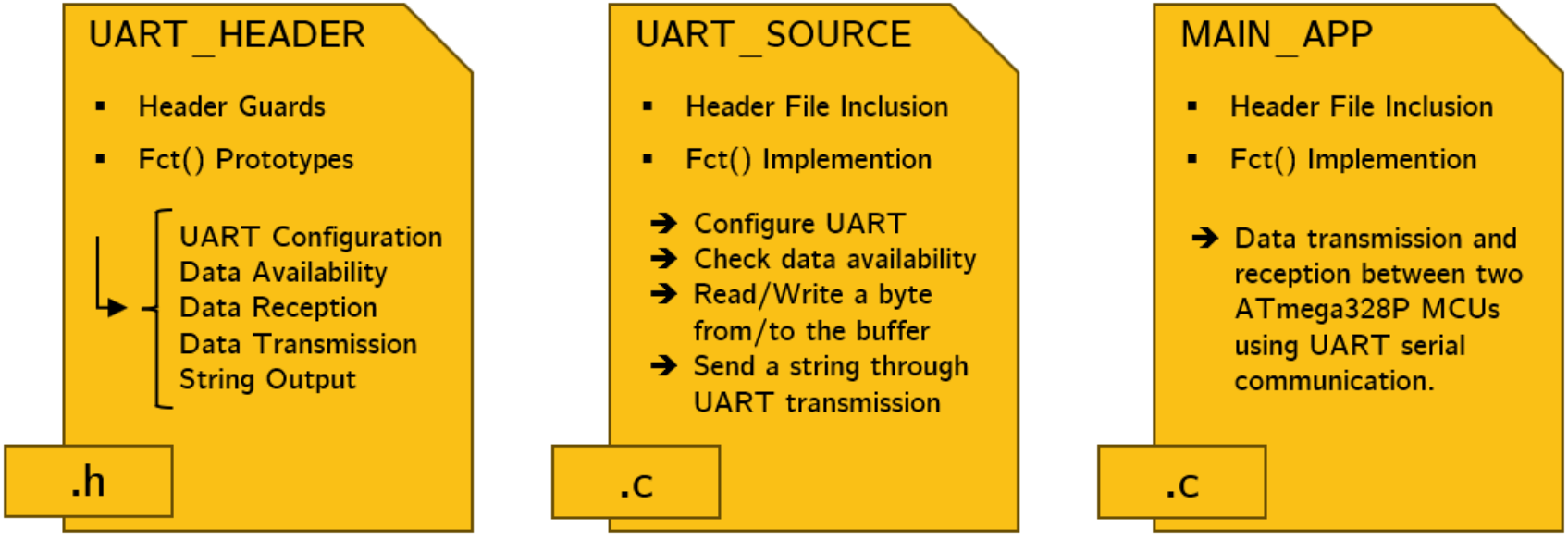
$$\mathbf{UBRR} = \frac{\mathbf{F}_{\mathrm{CPU}}}{\mathbf{16} \times \mathbf{Baud\ Rate}} - 1$$

- Regarding the frame format configuration, a common choice is 8 data

UART Driver Development

bits, no parity, and 1 stop bit because it balances simplicity, efficiency and compatibility. However, optimal depend on the application, and our driver will offer dynamic selection during initialization for flexibility.

- The illustration below presents a detailed overview of the key elements and structure of the driver files :



UART Driver Development

Let's bridge devices with UART communication!

1. Make sure the UART settings are identical between the transmitter and receiver. Mismatched settings will cause data transfer failure.
2. Generate the **.hex** files for both the transmitter and receiver using the commands from slide 15 with minor adjustments.
3. Load the **.hex** files onto the respective ATmega328P devices. A ready to use ISIS schematic is available on GitHub to help with driver testing.
4. Run the application and observe the system. The receiver's PORTB pins will display the received byte and sends **"Next"** to prompt the transmitter, which toggles the PORTB5 LED before sending the next byte.

§ Refer to the GitHub link below for the driver files and ISIS schematic : https://tinyurl.com/w6tzhaer

Thanks!

Contact:

Hamza Taous

E-mail: hamzataous847@gmail.com

Automotive SW Test & Validation Engineer

Stellantis on behalf of Capgemini Engineering

Casablanca, Morocco