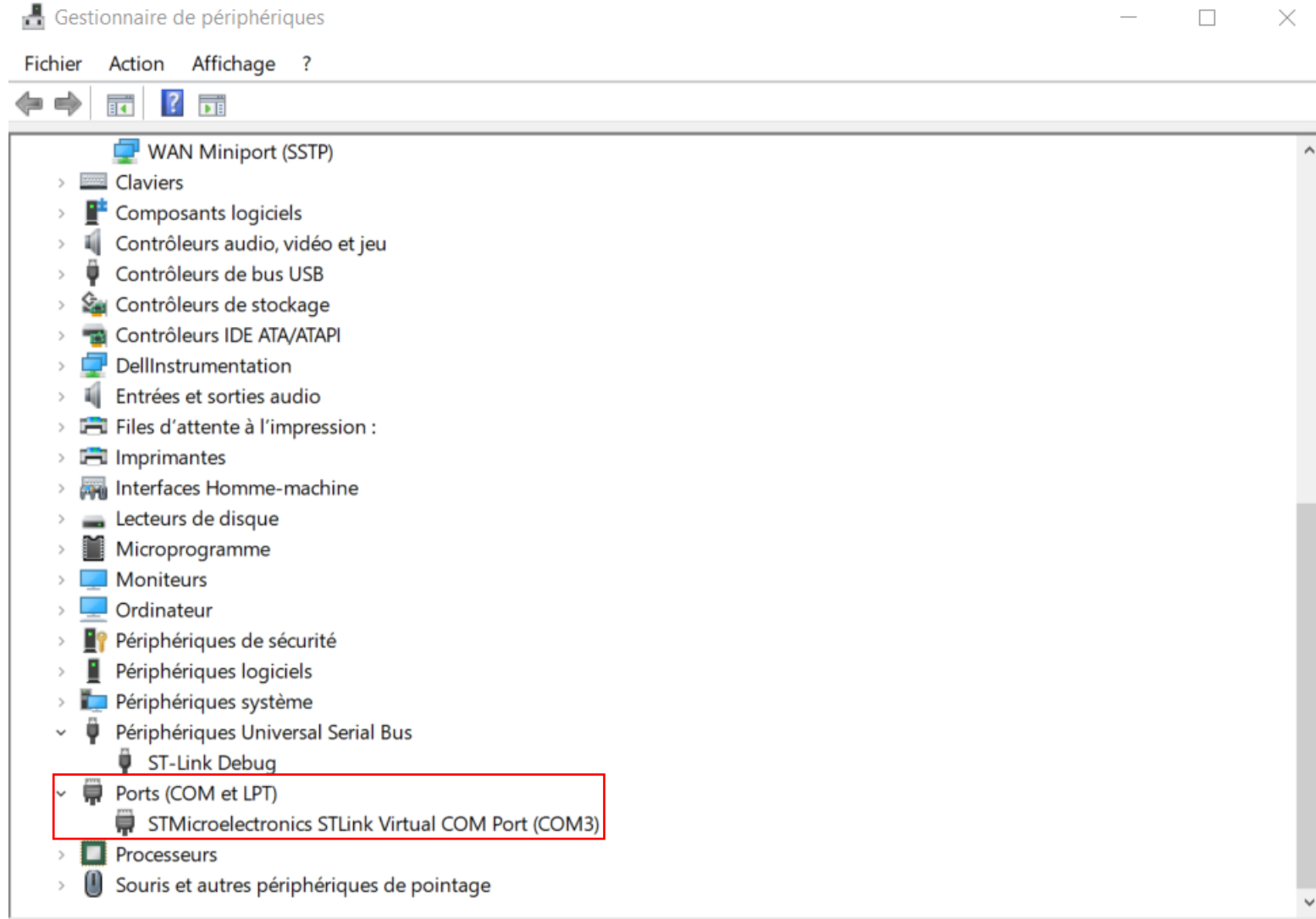


Learning-ADC_TO_UART

Ce document résume mes observations suite à la programmation de la carte de développement : « Nucleo-L476RG ».

L'objectif est d'enregistrer un signal en entrée de l'ADC et lire le buffer en utilisant la liaison UART.



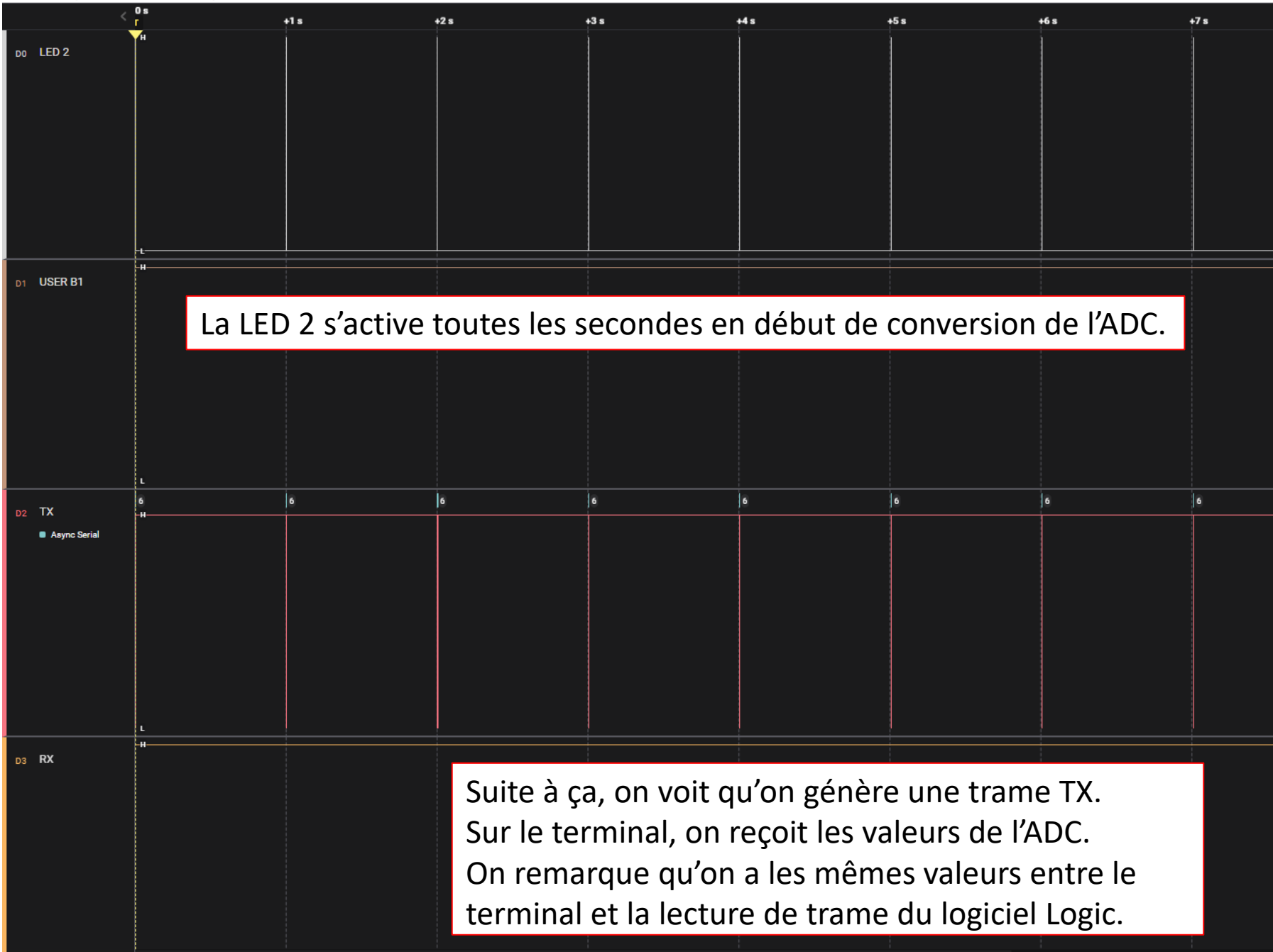
J'ai utilisé la liaison série accessible via le COM3 du STLink.

Première étape

On travail avec un ADC et une liaison UART dans la boucle « While ».

On répète ces actions toutes les secondes :

- 1) On fait une conversion ADC.
- 2) On a une valeur sur 12 bits (0 à 4094).
- 3) On passe cette valeur dans un tableau chiffre par chiffre en « char ».
- 4) On transmet ce buffer à l'UART.
- 5) L'UART transmet via TX le buffer.
- 6) On lit la valeur sur le terminal.



La LED 2 s'active toutes les secondes en début de conversion de l'ADC.

Suite à ça, on voit qu'on génère une trame TX.
Sur le terminal, on reçoit les valeurs de l'ADC.
On remarque qu'on a les mêmes valeurs entre le
terminal et la lecture de trame du logiciel Logic.

Analyzers

- Async Serial ✓

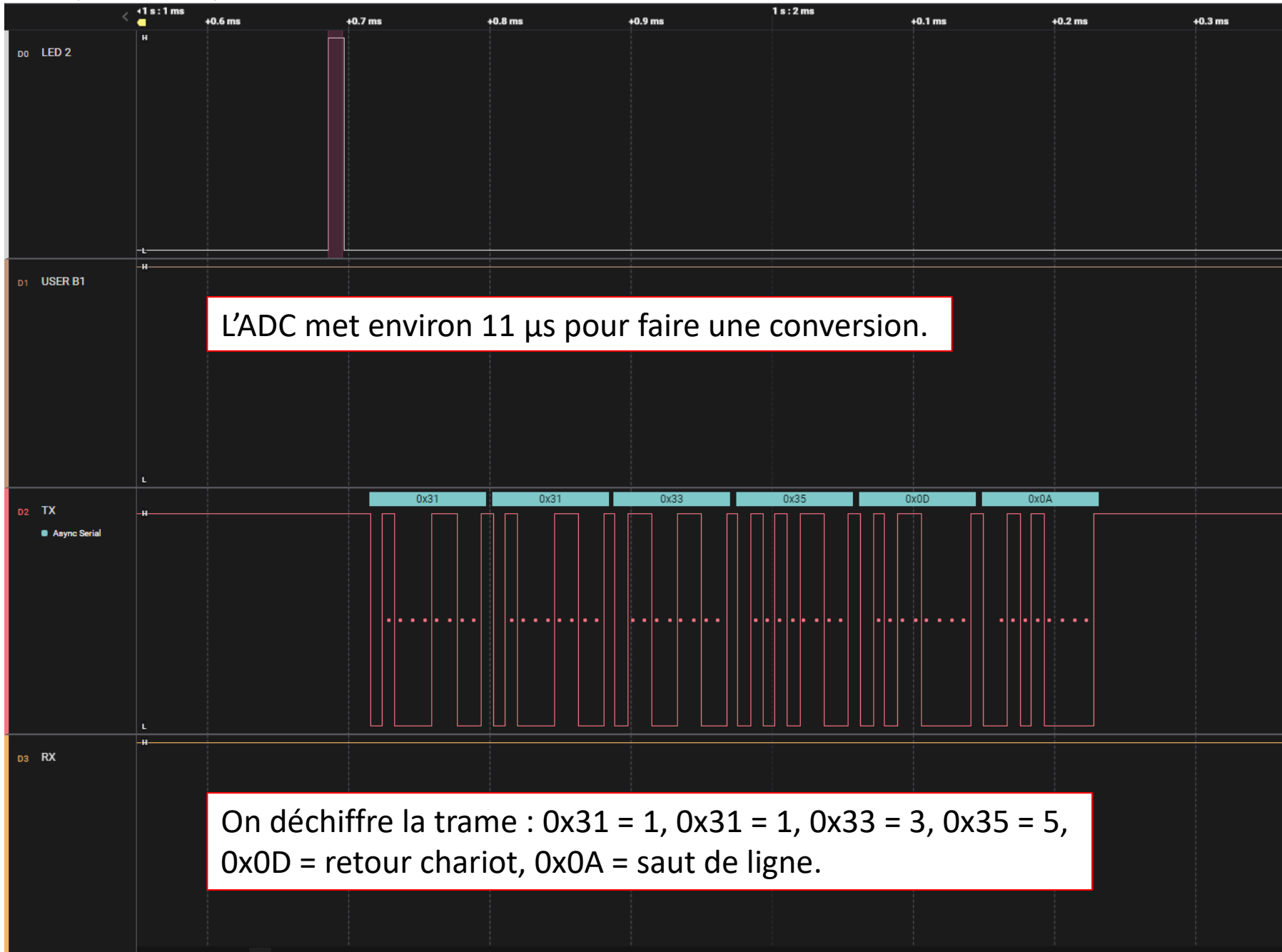
> Trigger View ▲

Data ? ✓

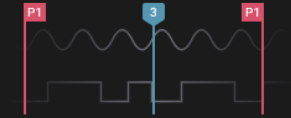
```
1137
1135
1134
1136
1145
1137
1142
1142
1136
1142
1143
```

COM3 - PuTTY

```
1141
1143
1141
1142
1139
1142
1137
1135
1134
1136
1145
1137
1142
1142
1136
1142
1143
1140
1134
1133
1144
1135
1140
```



Timing Markers



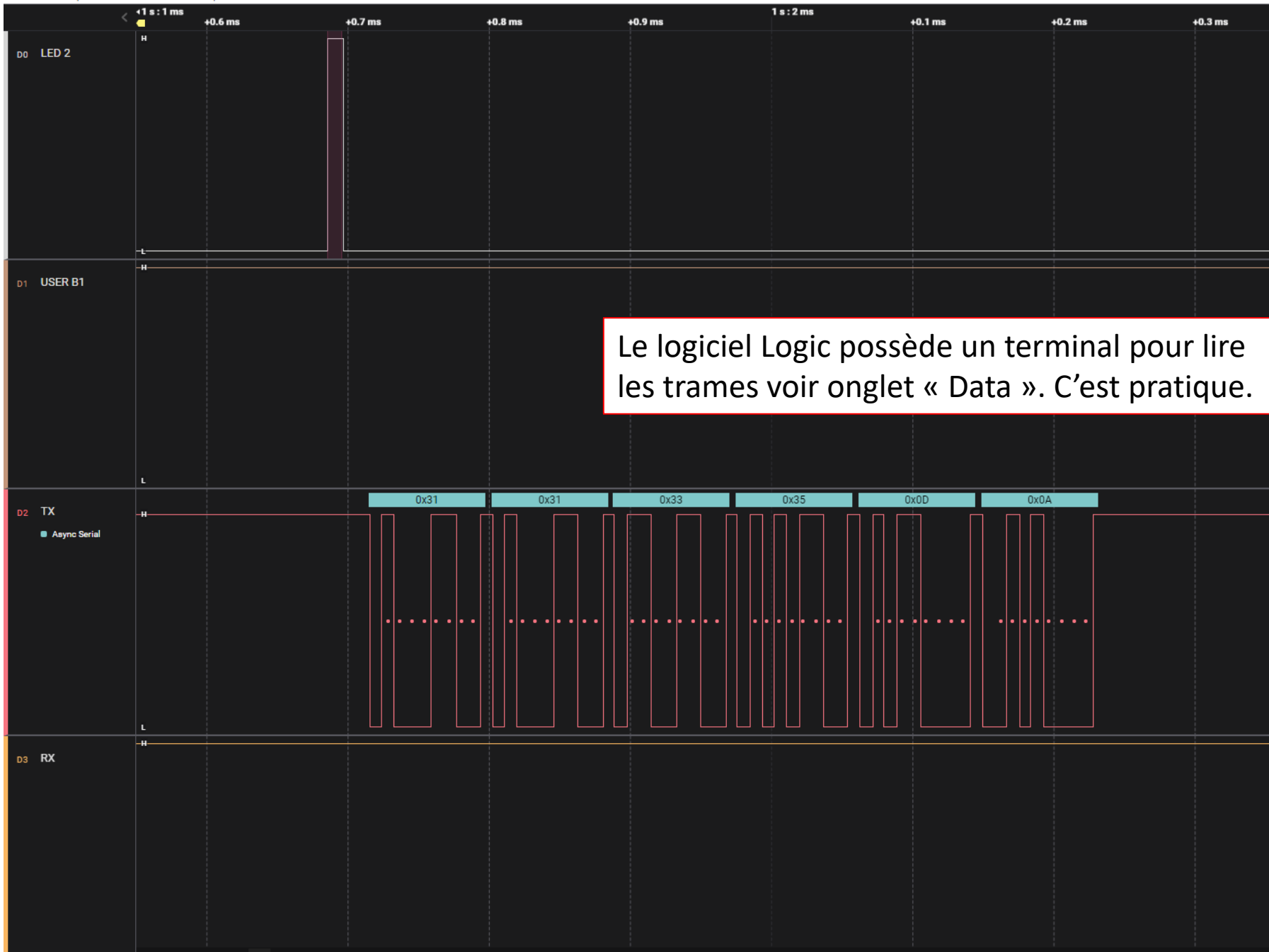
Click "+" to add Timing Markers

Measurements

M0	Δ 11 μ s
N _{falling}	1
N _{rising}	1
f _{min}	N/A
f _{max}	N/A
f _{mean}	N/A
T _{std}	N/A

Notes

Click to add a description of your capture



Le logiciel Logic possède un terminal pour lire les trames voir onglet « Data ». C'est pratique.

Analyzers

■ Async Serial ✓

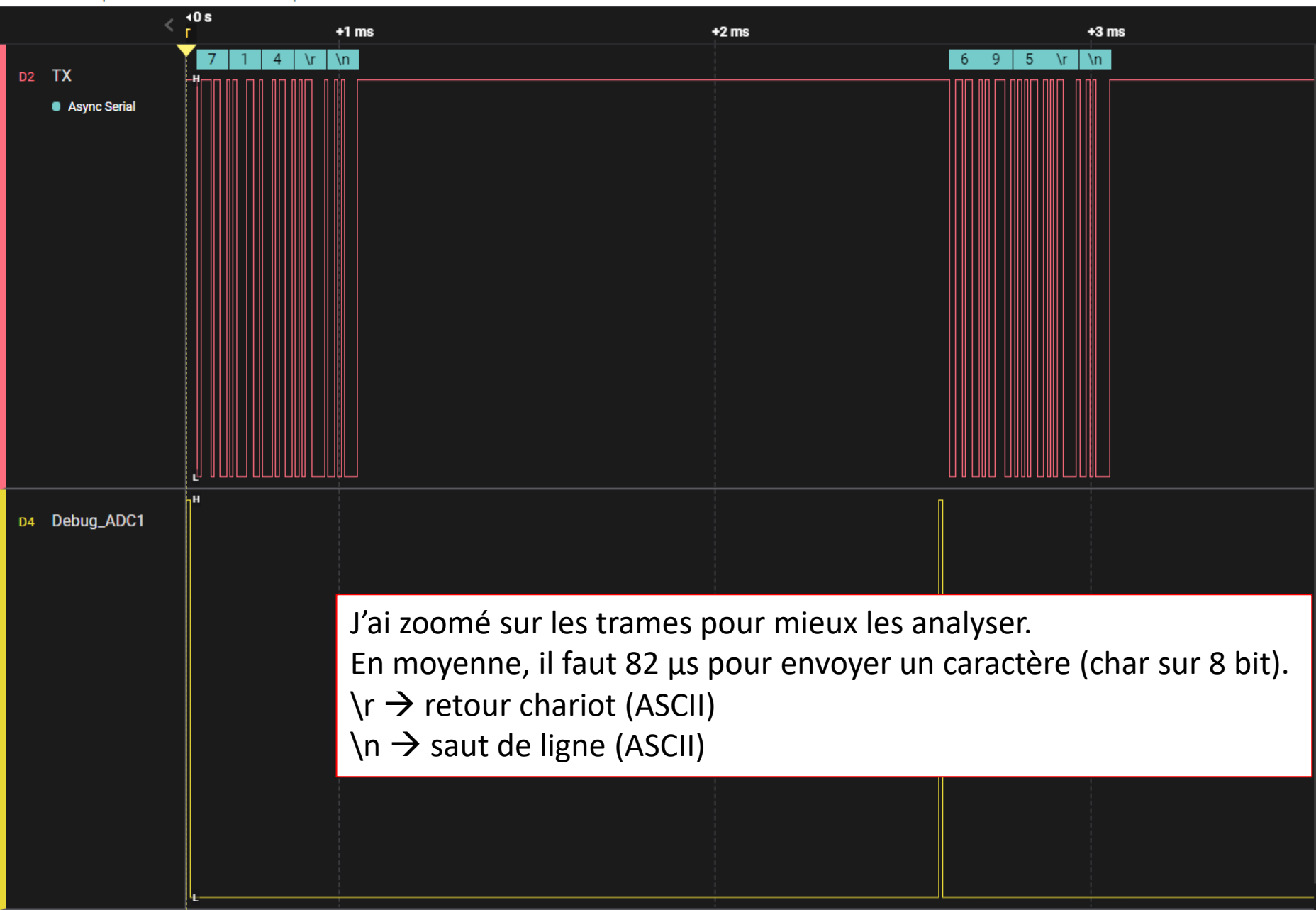
> Trigger View ▲

Data ? ✓

1137
1135
1134
1136
1145
1137
1142
1142
1136
1142
1143

```
COM3 - PuTTY
1132
1144
1140
1137
1141
1146
1129
1167
1168
  1147
    1133
      1136
        1135
          1131
            1134
              1144
                1134
                  1141
                    1134
                      1135
                        1135
                          1134
                            1137
```

J'ai supprimé le retour chariot pour voir son impact.



J'ai zoomé sur les trames pour mieux les analyser.
En moyenne, il faut 82 μ s pour envoyer un caractère (char sur 8 bit).
\r → retour chariot (ASCII)
\n → saut de ligne (ASCII)

Analyzers

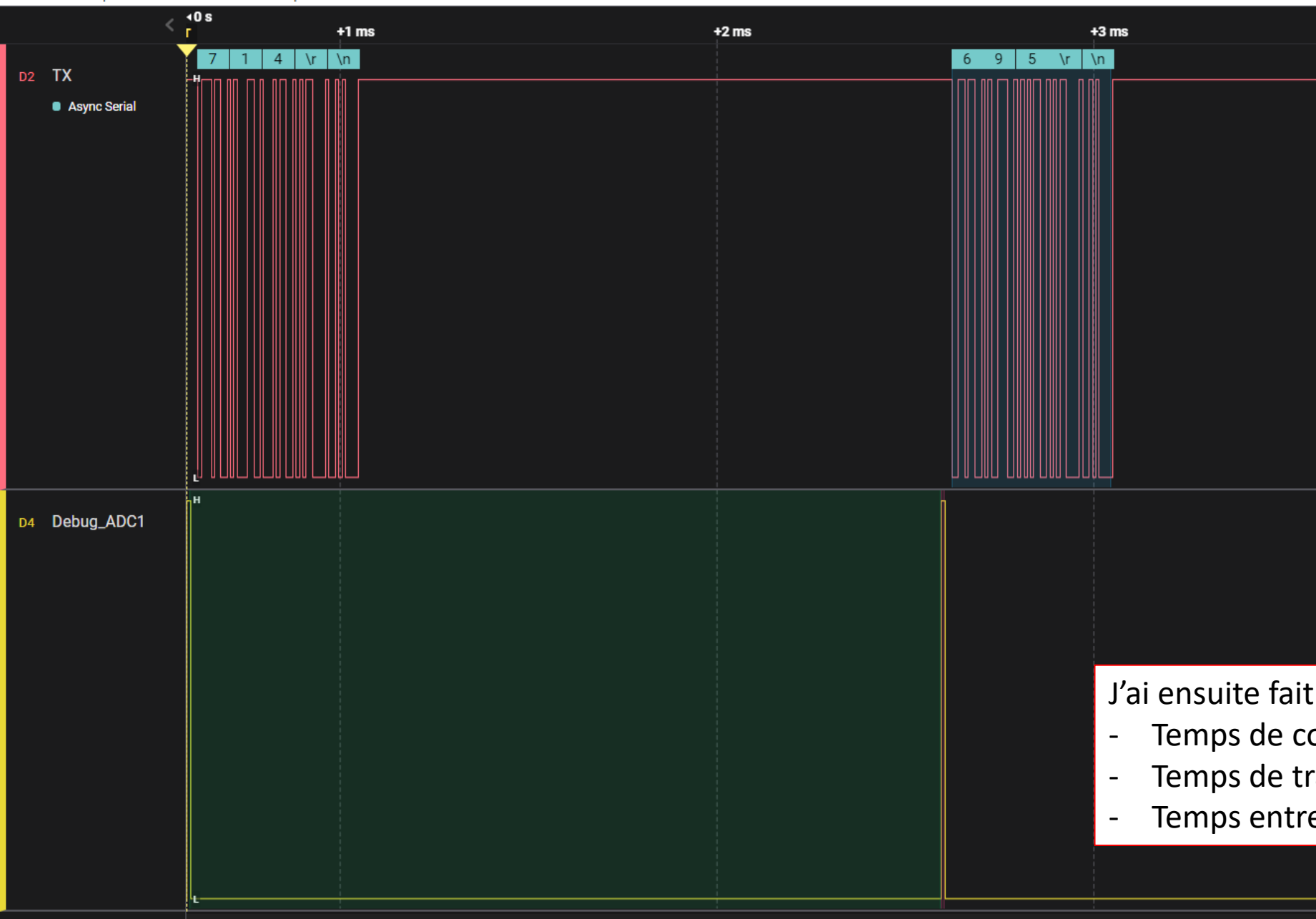
■ Async Serial ✓

> Trigger View ⚠

Data ? ✓

Type to search

	Type	Start	Duration	data	
■	data	622 μ s	82 μ s	7	
■	data	709 μ s	82 μ s	1	
■	data	796 μ s	82 μ s	4	
■	data	882 μ s	82 μ s	\r	
■	data	969 μ s	82 μ s	\n	
■	data	2.623 ms	82 μ s	6	
■	data	2.709 ms	82 μ s	9	
■	data	2.796 ms	82 μ s	5	
■	data	2.883 ms	82 μ s	\r	
■	data	2.97 ms	82 μ s	\n	



Timing Markers ?

Measurements ?

M0	→	Δ10 μs
M1	→	Δ1.991 ms
M2	→	Δ425 μs
N _{falling}		16
N _{rising}		16
f _{min}		22.727 kHz
f _{max}		58.824 kHz
f _{mean}		38.462 kHz
T _{std}		8.652 μs

Notes ?

The conversion time of the ADC1 is equal to 10μs.
The waiting time before a new conversion is about 2 ms.
5 Ascii characters are transmitted in 425 μs.

J'ai ensuite fait des mesures :

- Temps de conversion ADC = 10 μs
- Temps de transmission UART = 425 μs
- Temps entre chaque conversion = 2 ms

Première étape (Version 2)

On travail avec un ADC et une liaison UART dans la boucle « While ».

On répète ces actions toutes les secondes :

- 1) On fait une conversion ADC.
- 2) On a une valeur sur 12 bits (0 à 4094).
- 3) On passe cette valeur dans un tableau chiffre par chiffre en « char ».
- 4) On transmet ce buffer à l'UART.
- 5) L'UART transmet via TX le buffer.
- 6) On lit la valeur sur le terminal.

Dans cette version, on stock les 5 dernières valeurs périodiquement pour afficher la moyenne suite à 5 conversion de l'ADC. J'ai ajouté ce bout de code pour avoir une meilleur lisibilité du capteur. J'ai remarqué que les valeurs varient de +/- 20. La moyenne permet de lisser cette variation.

```
ADC values n°2 = 1956
ADC values n°3 = 1926
ADC values n°4 = 1947
ADC values n°5 = 1956
Average of the last 5 elements = 1950
```

```
ADC values n°1 = 1939
ADC values n°2 = 1967
ADC values n°3 = 1941
ADC values n°4 = 1975
ADC values n°5 = 1958
Average of the last 5 elements = 1956
```

```
ADC values n°1 = 1941
ADC values n°2 = 1926
ADC values n°3 = 1943
ADC values n°4 = 1931
ADC values n°5 = 1934
Average of the last 5 elements = 1935
```

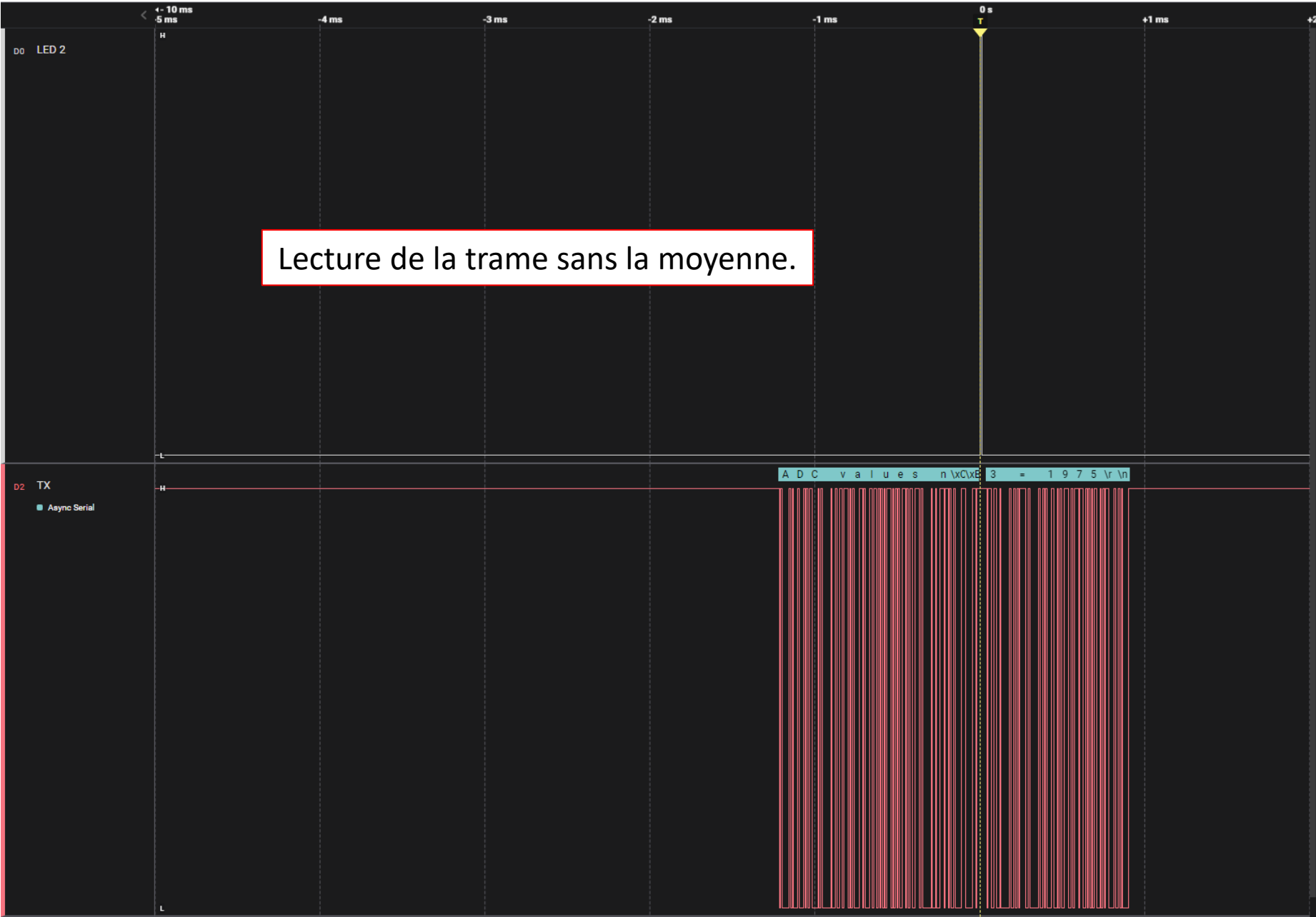
```
ADC values n°1 = 1967
ADC values n°2 = 1954
ADC values n°3 = 1960
ADC values n°4 = 1946
ADC values n°5 = 1960
Average of the last 5 elements = 1957
```

```
ADC values n°1 = 1956
ADC values n°2 = 1933
ADC values n°3 = 1963
ADC values n°4 = 1955
ADC values n°5 = 1935
Average of the last 5 elements = 1948
```

```
ADC values n°1 = 1941
ADC values n°2 = 1966
ADC values n°3 = 1956
ADC values n°4 = 1973
ADC values n°5 = 1960
Average of the last 5 elements = 1959
```

```
ADC values n°1 = 1926
ADC values n°2 = 1971
ADC values n°3 = 1947
ADC values n°4 = 1972
ADC values n°5 = 1923
Average of the last 5 elements = 1947
```

Voici le nouvel affichage.
Toutes les 5 valeurs, je fais une moyenne.

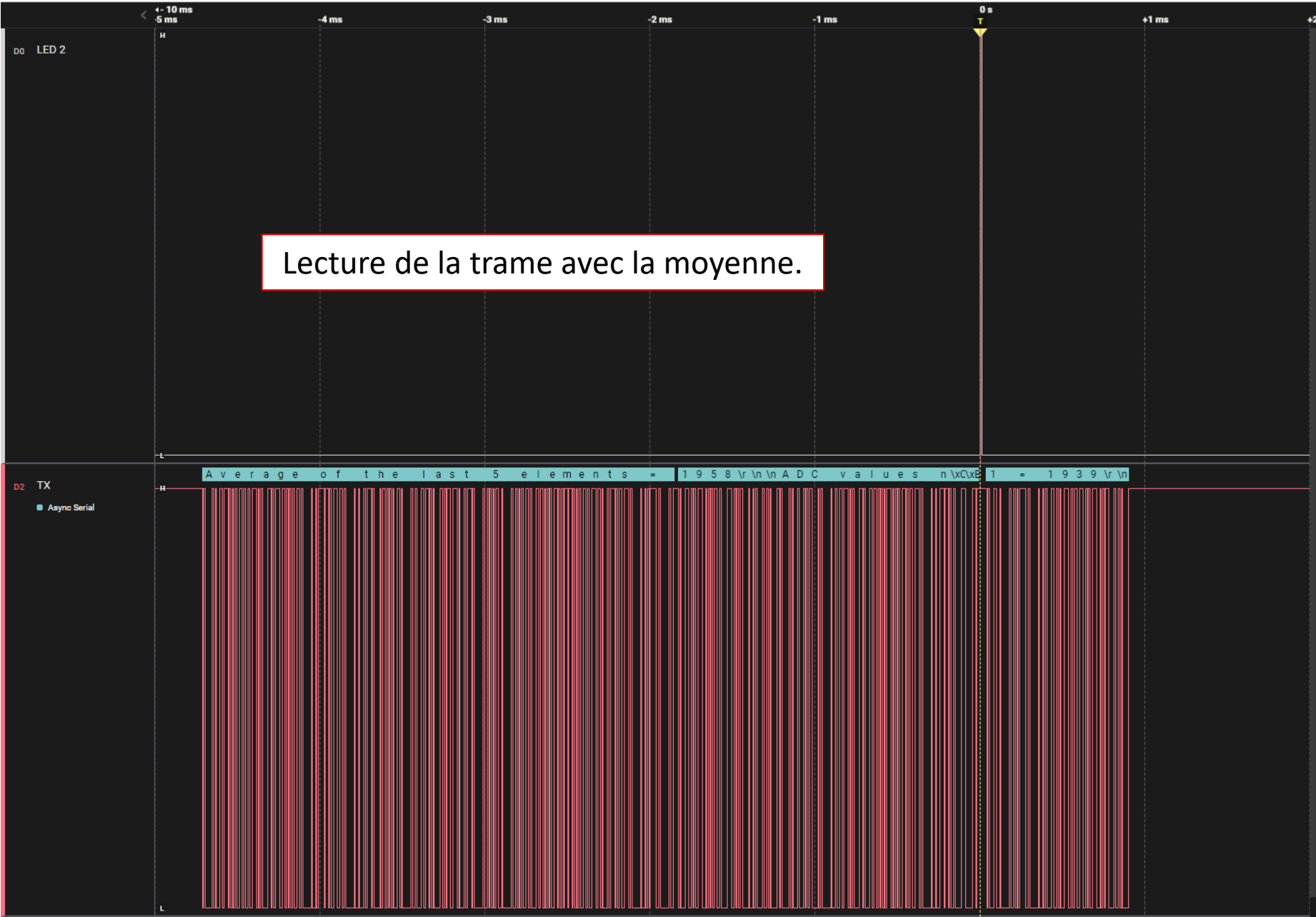


Timing Markers ? +

Measurements ? +

Notes ?

Test Code "Learning-TIMER_DMA"
First Step: use UART to read ADC values in while loop (version 1)



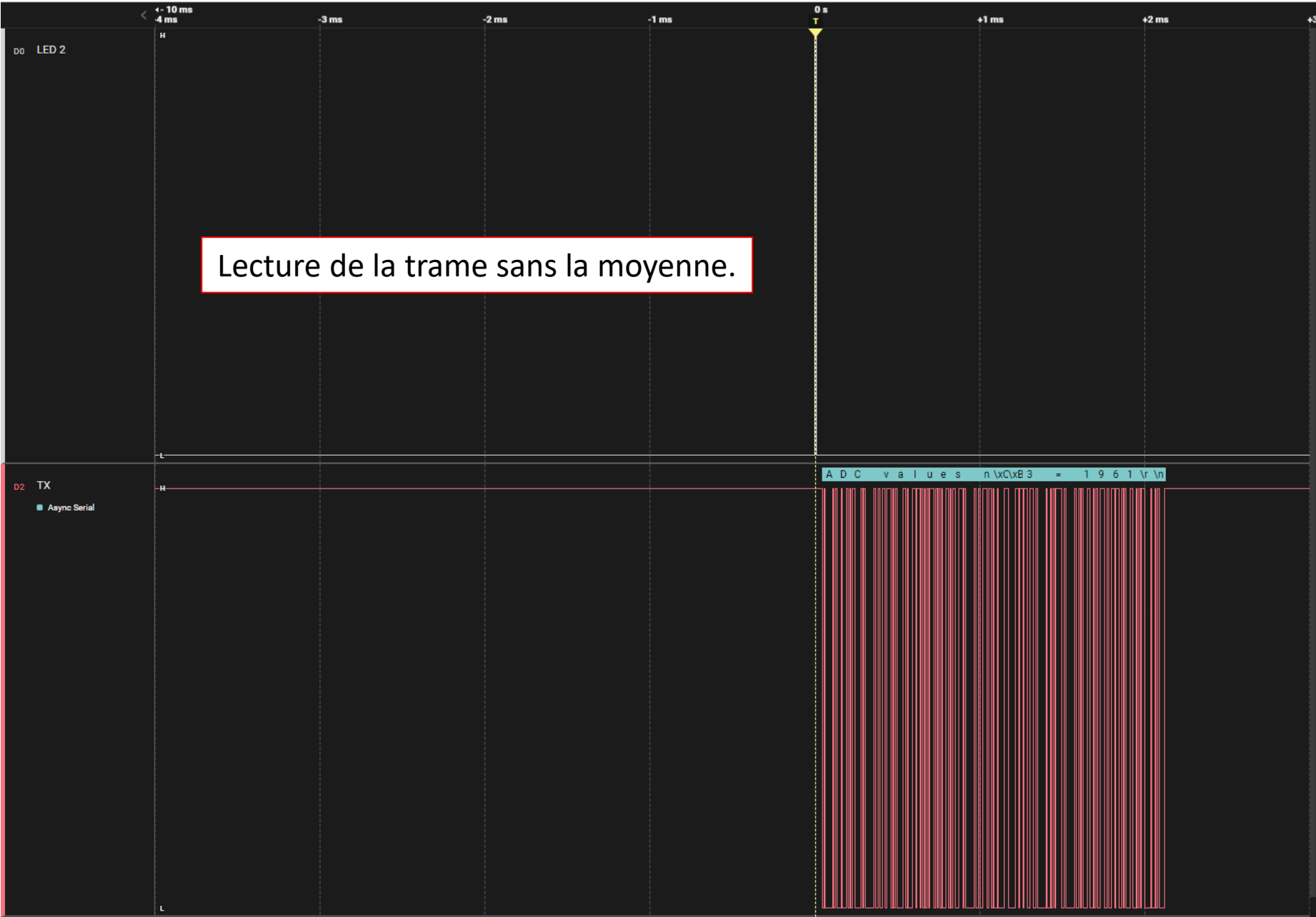
Timing Markers

Measurements

M0	→	Δ10 μs
N _{falling}		1
N _{rising}		1
f _{min}		N/A
f _{max}		N/A
f _{mean}		N/A
T _{std}		N/A
D _{cycle}		N/A

Notes

Test Code "Learning-TIMER_DMA"
First Step: use UART to read ADC values in while loop (version 1)



Timing Markers ? +

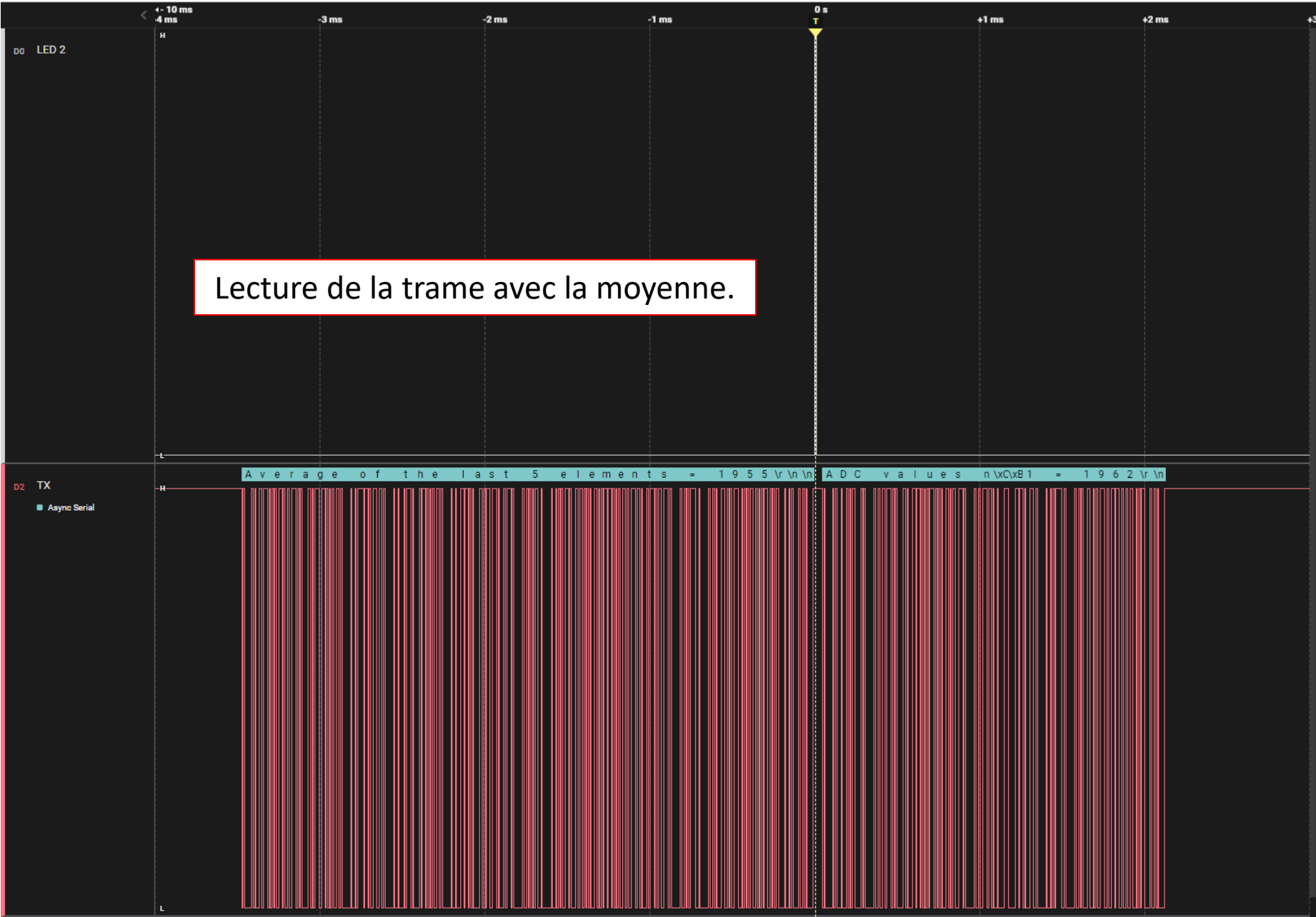
Measurements ? +

Notes ?

Test Code "Learning-TIMER_DMA"
First Step: use UART to read ADC values in while loop (version 2)

520 μs ^

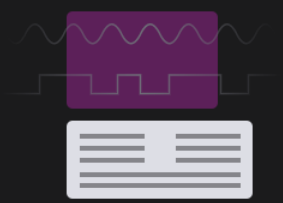
15:21
30/05/2023



Lecture de la trame avec la moyenne.

Timing Markers ?

Measurements ?



Notes ?

Test Code "Learning-TIMER_DMA"
First Step: use UART to read ADC values in
while loop (version 2)

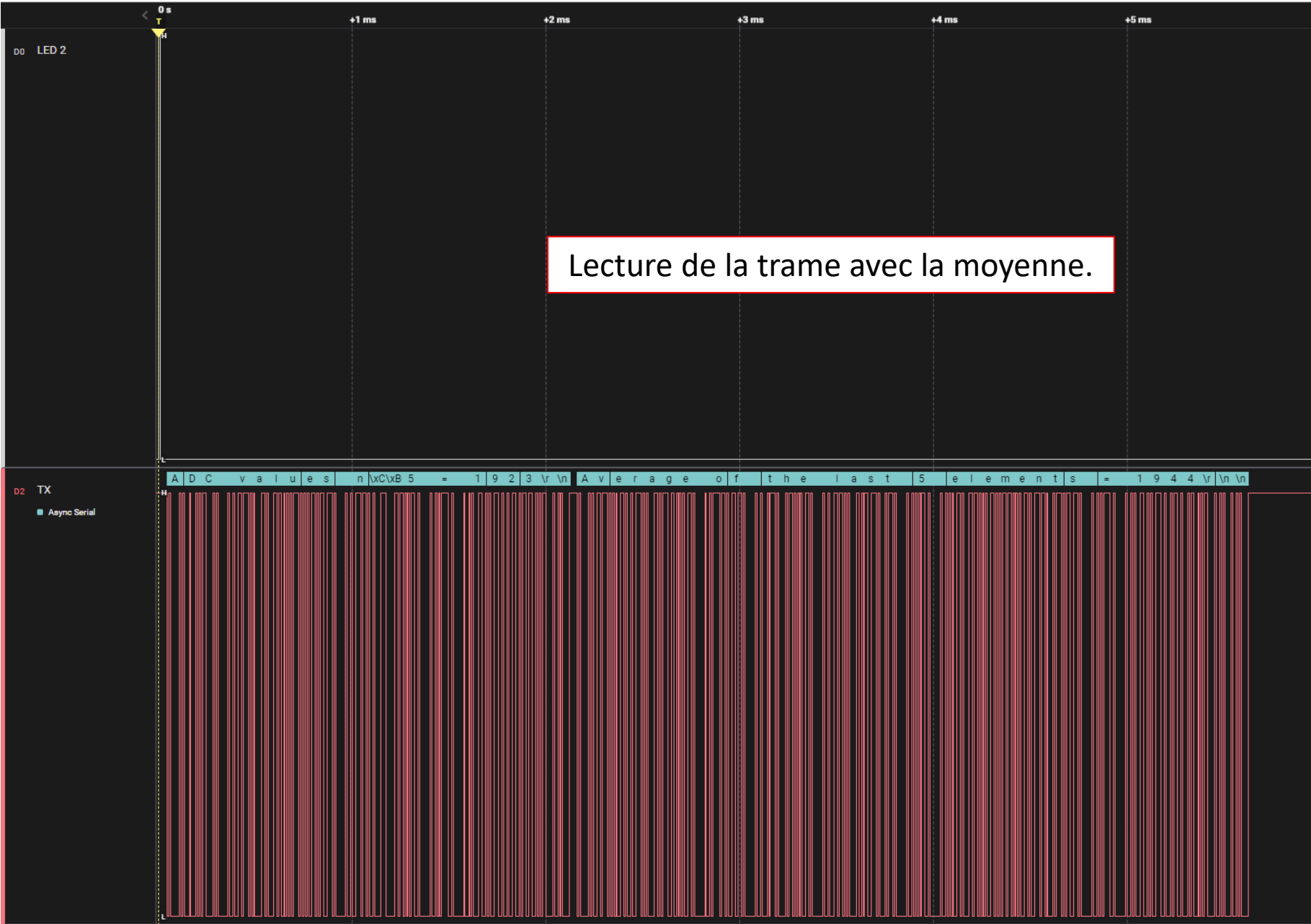


Timing Markers ? +

Measurements ? +

Notes ?

Test Code "Learning-TIMER_DMA"
First Step: use UART to read ADC values in while loop (version 3)



Lecture de la trame avec la moyenne.

Timing Markers ? +

Measurements ? +

Notes ?

Test Code "Learning-TIMER_DMA"
First Step: use UART to read ADC values in while loop (version 3)

446 μs ^

15:27
30/05/2023

Deuxième étape

J'utilise une DMA sur la transmission de l'UART pour envoyer un buffer avec beaucoup de valeur.

L'objectif de l'utilisation de la DMA dans ce cas est de ne pas monopoliser le CPU pour transmettre un grand buffer.

C'est une première utilisation de la DMA pour moi.

J'ai créé un tableau avec un long message et dans la boucle « While », j'envoie ce message toutes les secondes.

Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

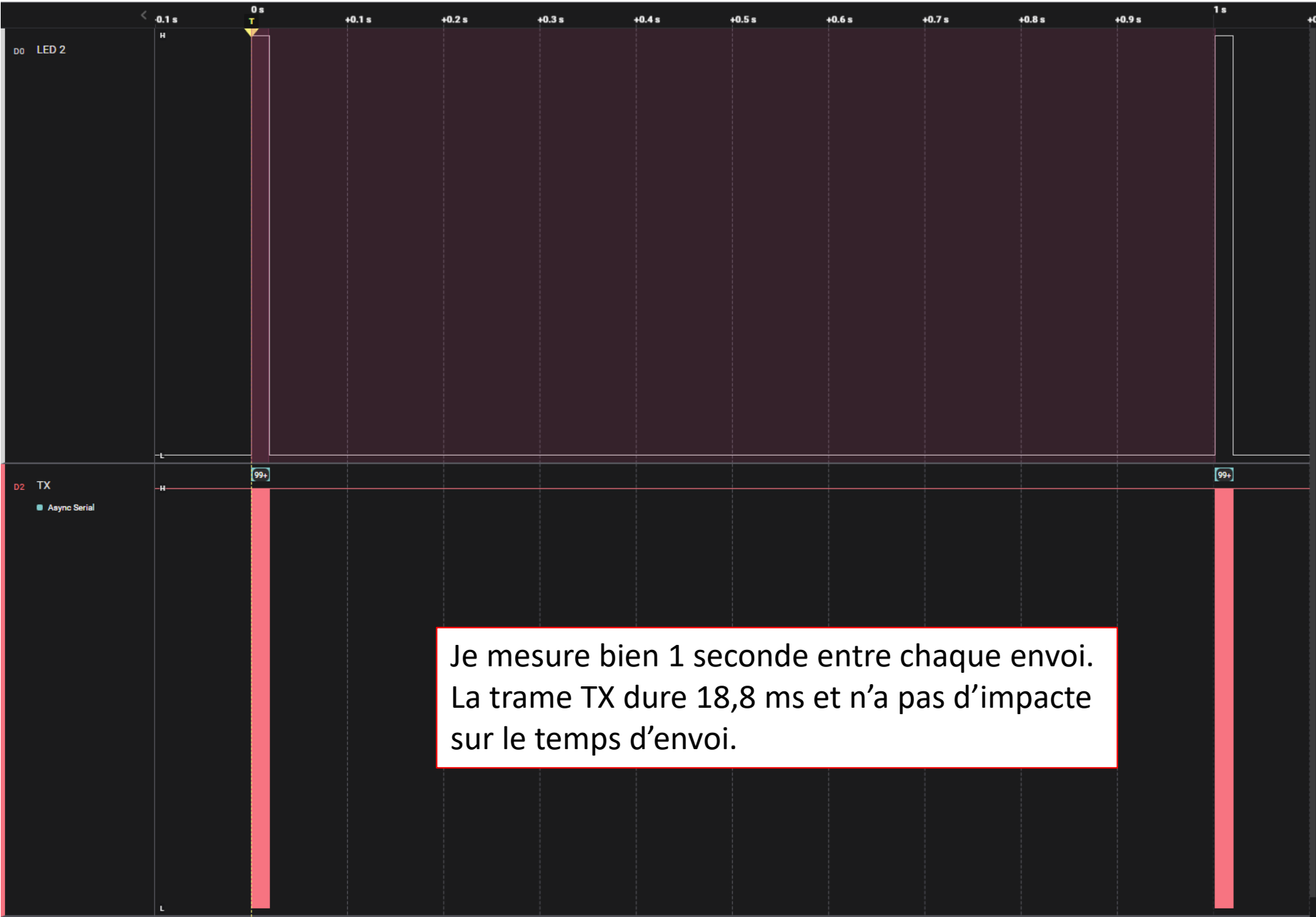
Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Ce message est un long message pour le test du DMA1.
Nous utilisons le DMA1 sur le périphérique USART2 TX.
L'objectif est de faire un code simple qui explicite
le fonctionnement du DMA1 pour un novice comme moi.

Voici l'affichage du long message
sur le terminal en liaison UART.



Timing Markers

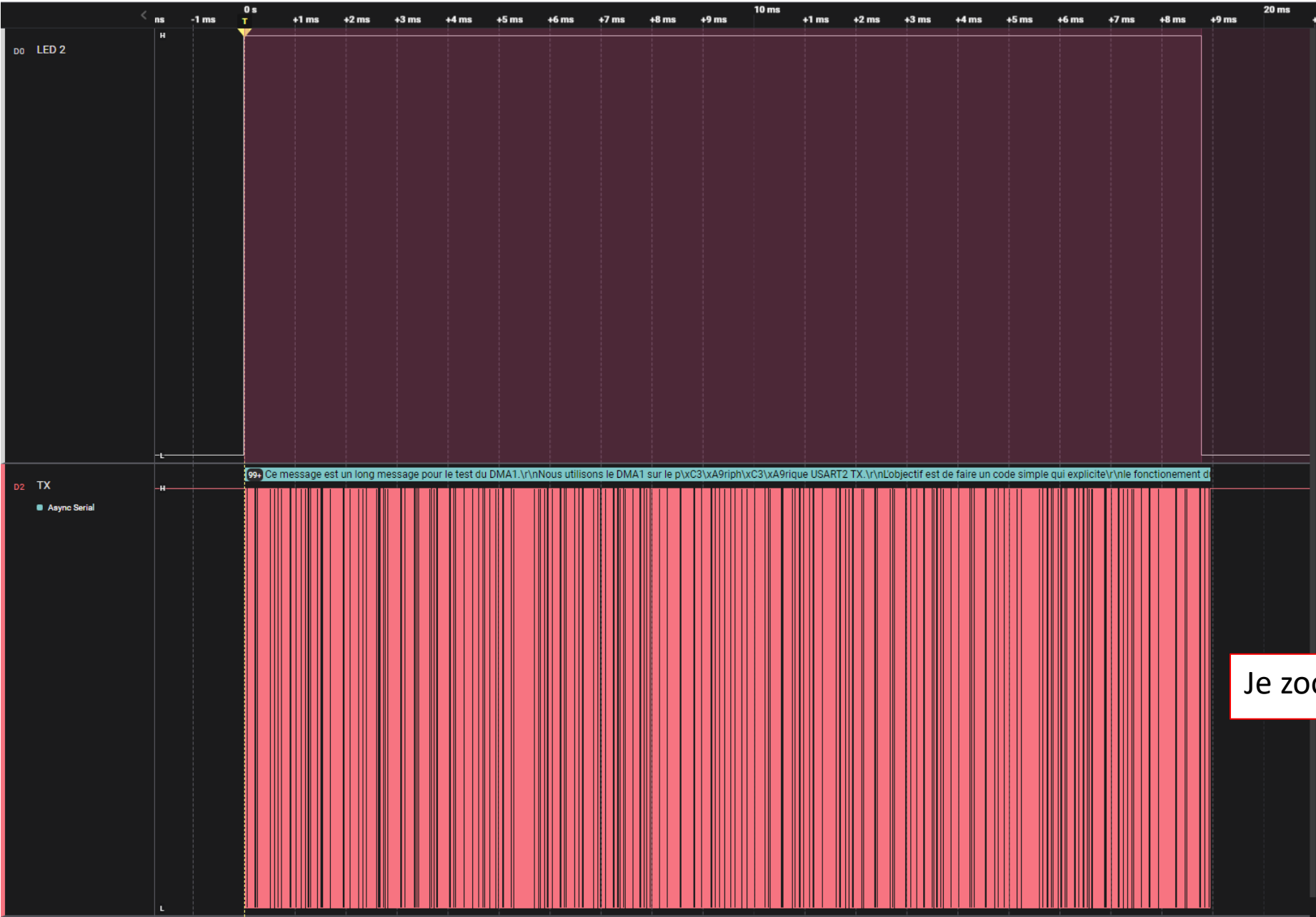
Measurements

M0	→	Δ18.777 ms
N _{falling}		1
N _{rising}		1
f _{min}		N/A
f _{max}		N/A
f _{mean}		N/A
T _{std}		N/A
D _{cycle}		N/A
M1	→	Δ1.001 755 000 s
N _{falling}		1
N _{rising}		2
f _{min}		N/A
f _{max}		998.248 mHz
f _{mean}		998.248 mHz
T _{std}		N/A
D _{cycle}		1.874 %

Notes

Test Code "Learning-TIMER_DMA"
Second Step: Using DMA1 over USART2 TX to transmit a long test message

Je mesure bien 1 seconde entre chaque envoi.
La trame TX dure 18,8 ms et n'a pas d'impacte
sur le temps d'envoi.



Timing Markers

Measurements

M0	→	Δ18.777 ms
N _{falling}		1
N _{rising}		1
f _{min}		N/A
f _{max}		N/A
f _{mean}		N/A
T _{std}		N/A
D _{cycle}		N/A
M1	→	Δ1.001 755 000 s
N _{falling}		1
N _{rising}		2
f _{min}		N/A
f _{max}		998.248 mHz
f _{mean}		998.248 mHz
T _{std}		N/A
D _{cycle}		1.874 %

Notes

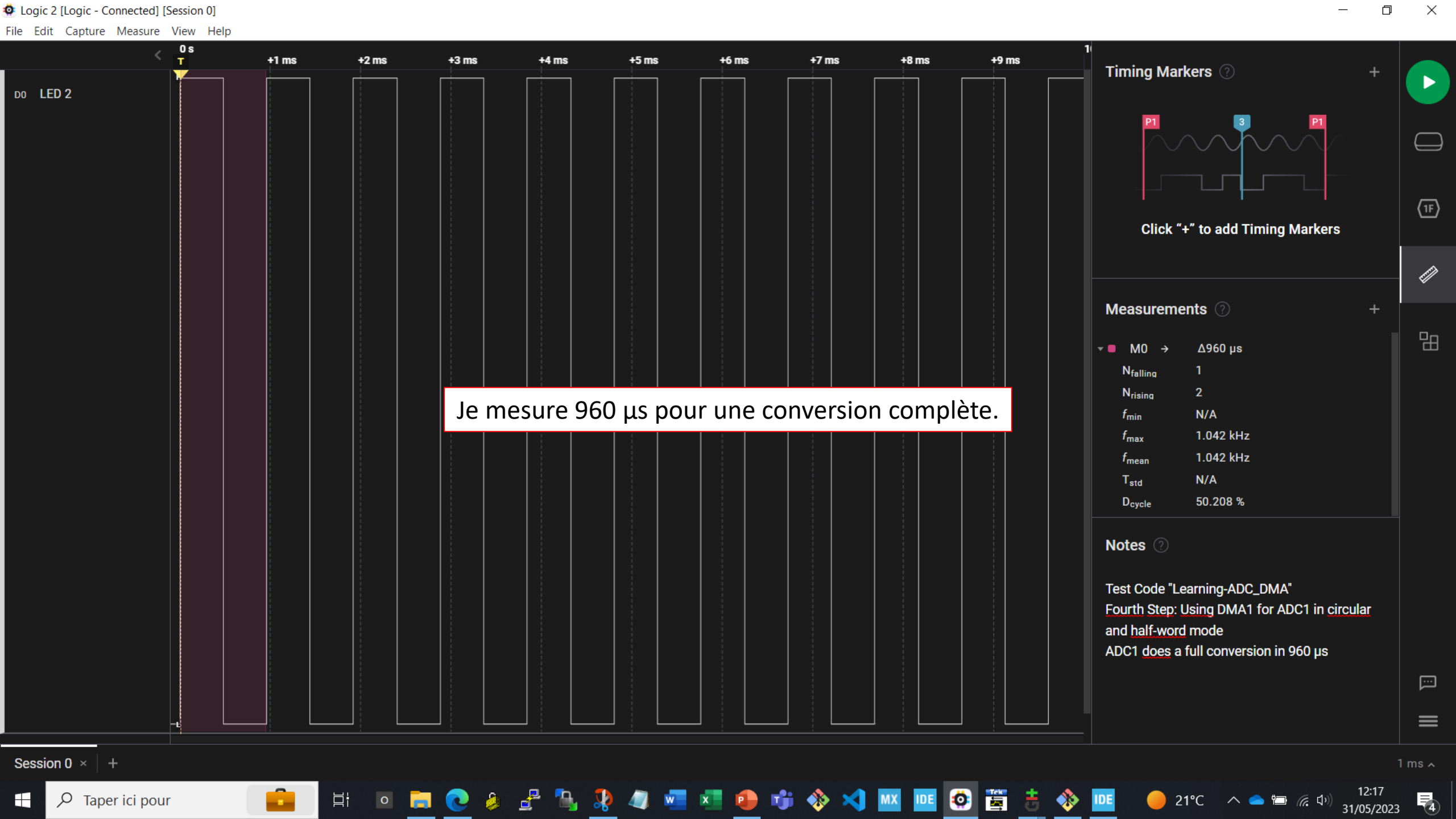
Test Code "Learning-TIMER_DMA"
Second Step: [Using DMA1 over USART2 TX](#) to transmit a long test message

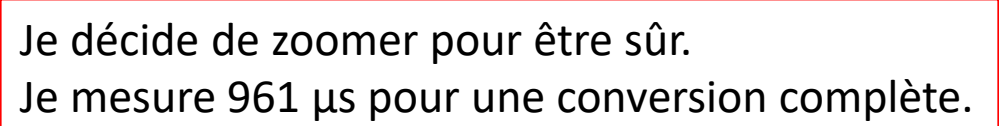
Je zoom sur la Trame TX.

Troisième étape

Cette fois-ci, j'utilise la DMA sur l'ADC en mode circulaire pour voir son impacte sur le débogueur et mesurer le temps de conversion sur un buffer.

On crée un buffer de 4096 valeurs. L'ADC remplit le buffer de manière cyclique et en parallèle du CPU grâce à la DMA.





main.c

Lib_AdrEnc

main.c ×

startup_stm321476rgtxs

```

312 /* USER CODE BEGIN SysInit */
313 /* USER CODE END SysInit */
314
315 /* Initialize all configured peripherals */
316 MX_GPIO_Init();
317 MX_DMA_Init();
318 MX_USART2_UART_Init();
319 MX_ADC1_Init();
320
321 /* USER CODE BEGIN 2 */
322 HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buf, ADC_BUF_LEN);
323 /* USER CODE END 2 */
324
325 /* Infinite loop */
326 /* USER CODE BEGIN WHILE */
327 while (1)
328 {
329 /* USER CODE END WHILE */
330
331 /* USER CODE BEGIN 3 */
332
333 }
334 /* USER CODE END 3 */
335
336 /**
337 * @brief System Clock Configuration
338 * @retval None
339 */
340 void SystemClock_Config(void)
341 {
342 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
343 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

```

Expression	Type	Value
adc_buf[0]	uint16_t	0
adc_buf[1]	uint16_t	0
adc_buf[2]	uint16_t	0
adc_buf[3]	uint16_t	0
adc_buf[4]	uint16_t	0
adc_buf[5]	uint16_t	0
adc_buf[6]	uint16_t	0
adc_buf[7]	uint16_t	0
adc_buf[8]	uint16_t	0
adc_buf[9]	uint16_t	0
adc_buf[10]	uint16_t	0
adc_buf[11]	uint16_t	0
adc_buf[12]	uint16_t	0
adc_buf[13]	uint16_t	0
adc_buf[14]	uint16_t	0
adc_buf[15]	uint16_t	0
adc_buf[16]	uint16_t	0
adc_buf[17]	uint16_t	0
adc_buf[18]	uint16_t	0
adc_buf[19]	uint16_t	0
adc_buf[20]	uint16_t	0
adc_buf[21]	uint16_t	0
adc_buf[22]	uint16_t	0
adc_buf[23]	uint16_t	0

main.c

Lib_AdrEnc

main.c ×

startup_stm321476rgtxs

```

312 /* USER CODE BEGIN MX_GPIO_Init_2 */
313 /* USER CODE END MX_GPIO_Init_2 */
314
315 /* USER CODE BEGIN 4 */
316 // Called when first half of buffer is filled
317 void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc)
318 {
319 // Set LD2
320 HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
321
322 // Called when buffer is completely filled
323 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
324 {
325 // Reset LD2
326 HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
327
328 /* USER CODE END 4 */
329
330 /**
331 * @brief This function is executed in case of error occurrence.
332 * @retval None
333 */
334 void Error_Handler(void)
335 {
336 /* User can add his own implementation to report the HAL error return state */
337 __disable_irq();
338 while (1)
339 {
340
341 }
342
343 }

```

Expression	Type	Value
adc_buf[0]	uint16_t	1975
adc_buf[1]	uint16_t	1964
adc_buf[2]	uint16_t	1943
adc_buf[3]	uint16_t	1946
adc_buf[4]	uint16_t	1964
adc_buf[5]	uint16_t	1942
adc_buf[6]	uint16_t	1978
adc_buf[7]	uint16_t	1966
adc_buf[8]	uint16_t	1972
adc_buf[9]	uint16_t	1939
adc_buf[10]	uint16_t	1985
adc_buf[11]	uint16_t	1955
adc_buf[12]	uint16_t	1950
adc_buf[13]	uint16_t	1963
adc_buf[14]	uint16_t	1983
adc_buf[15]	uint16_t	1970
adc_buf[16]	uint16_t	1943
adc_buf[17]	uint16_t	1943
adc_buf[18]	uint16_t	1952
adc_buf[19]	uint16_t	1962
adc_buf[20]	uint16_t	1972
adc_buf[21]	uint16_t	1953
adc_buf[22]	uint16_t	1935
adc_buf[23]	uint16_t	1962

On passe en mode débuge pour voir le remplissage du buffer par la DMA.

main.c

Lib_AdrEnc

main.c ×

startup_stm321476rgtxs

```

312 /* USER CODE BEGIN MX_GPIO_Init_2 */
313 /* USER CODE END MX_GPIO_Init_2 */
314
315 /* USER CODE BEGIN 4 */
316 // Called when first half of buffer is filled
317 void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc)
318 {
319 // Set LD2
320 HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
321
322 // Called when buffer is completely filled
323 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
324 {
325 // Reset LD2
326 HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
327
328 /* USER CODE END 4 */
329
330 /**
331 * @brief This function is executed in case of error occurrence.
332 * @retval None
333 */
334 void Error_Handler(void)
335 {
336 /* User CODE BEGIN Error_Handler_Debug */
337 /* User can add his own implementation to report the HAL error return state */
338 __disable_irq();
339 while (1)
340 {
341
342 }
343
344 }

```

Expression	Type	Value
adc_buf[2800]	uint16_t	1964
adc_buf[2801]	uint16_t	1982
adc_buf[2802]	uint16_t	1947
adc_buf[2803]	uint16_t	1973
adc_buf[2804]	uint16_t	1950
adc_buf[2805]	uint16_t	1937
adc_buf[2806]	uint16_t	1944
adc_buf[2807]	uint16_t	1943
adc_buf[2808]	uint16_t	1943
adc_buf[2809]	uint16_t	1985
adc_buf[2810]	uint16_t	1937
adc_buf[2811]	uint16_t	1960
adc_buf[2812]	uint16_t	1966
adc_buf[2813]	uint16_t	1953
adc_buf[2814]	uint16_t	1983
adc_buf[2815]	uint16_t	1967
adc_buf[2816]	uint16_t	1970
adc_buf[2817]	uint16_t	1935
adc_buf[2818]	uint16_t	1983
adc_buf[2819]	uint16_t	1946
adc_buf[2820]	uint16_t	1966
adc_buf[2821]	uint16_t	1934

main.c

Lib_AdrEnc

main.c ×

startup_stm321476rgtxs

```

312 /* USER CODE BEGIN MX_GPIO_Init_2 */
313 /* USER CODE END MX_GPIO_Init_2 */
314
315 /* USER CODE BEGIN 4 */
316 // Called when first half of buffer is filled
317 void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc)
318 {
319 // Set LD2
320 HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
321
322 // Called when buffer is completely filled
323 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
324 {
325 // Reset LD2
326 HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
327
328 /* USER CODE END 4 */
329
330 /**
331 * @brief This function is executed in case of error occurrence.
332 * @retval None
333 */
334 void Error_Handler(void)
335 {
336 /* User CODE BEGIN Error_Handler_Debug */
337 /* User can add his own implementation to report the HAL error return state */
338 __disable_irq();
339 while (1)
340 {
341
342 }
343
344 }

```

Expression	Type	Value
adc_buf[4000]	uint16_t	1955
adc_buf[4001]	uint16_t	1951
adc_buf[4002]	uint16_t	1951
adc_buf[4003]	uint16_t	1978
adc_buf[4004]	uint16_t	1946
adc_buf[4005]	uint16_t	1963
adc_buf[4006]	uint16_t	1983
adc_buf[4007]	uint16_t	1944
adc_buf[4008]	uint16_t	1966
adc_buf[4009]	uint16_t	1978
adc_buf[4010]	uint16_t	1957
adc_buf[4011]	uint16_t	1982
adc_buf[4012]	uint16_t	1951
adc_buf[4013]	uint16_t	1963
adc_buf[4014]	uint16_t	1943
adc_buf[4015]	uint16_t	1965
adc_buf[4016]	uint16_t	1963
adc_buf[4017]	uint16_t	1951
adc_buf[4018]	uint16_t	1933
adc_buf[4019]	uint16_t	1977
adc_buf[4020]	uint16_t	1976
adc_buf[4021]	uint16_t	1967
adc_buf[4022]	uint16_t	1936
adc_buf[4023]	uint16_t	1952

main.c

Lib_Adrien.c

main.c

startup_stm321476rgtx.s

```

309  GPIO_InitStruct.Pull = GPIO_NOPULL;
310  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
311  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
312
313  /* USER CODE BEGIN MX_GPIO_Init_2 */
314  /* USER CODE END MX_GPIO_Init_2 */
315  }
316
317  /* USER CODE BEGIN 4 */
318  // Called when first half of buffer is filled
319  void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc)
320  {
321      // Set LD2
322      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
323  }
324
325  // Called when buffer is completely filled
326  void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
327  {
328      // Reset LD2
329      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
330  }
331  /* USER CODE END 4 */
332
333  /**
334   * @brief This function is executed in case of error occurrence.
335   * @retval None
336   */
337  void Error_Handler(void)
338  {
339      /* USER CODE BEGIN Error_Handler_Debug */
340      /* User can add his own implementation to report the HAL error return state */

```

Expr...

Regi...

Live ...

SFRs

Expression	Type	Value
adc_buf	uint16_t [40...	0x20000...
[0..99]	uint16_t [10...	0x20000...
adc_buf[0]	uint16_t	3083
adc_buf[1]	uint16_t	3092
adc_buf[2]	uint16_t	3078
adc_buf[3]	uint16_t	3078
adc_buf[4]	uint16_t	3084
adc_buf[5]	uint16_t	3072
adc_buf[6]	uint16_t	3077
adc_buf[7]	uint16_t	3067
adc_buf[8]	uint16_t	3066
adc_buf[9]	uint16_t	3039
adc_buf[10]	uint16_t	3053
adc_buf[11]	uint16_t	3060
adc_buf[12]	uint16_t	3053
adc_buf[13]	uint16_t	3084
adc_buf[14]	uint16_t	3079
adc_buf[15]	uint16_t	3059
adc_buf[16]	uint16_t	3084
adc_buf[17]	uint16_t	3064
adc_buf[18]	uint16_t	3072
adc_buf[19]	uint16_t	3063
adc_buf[20]	uint16_t	3075
adc_buf[21]	uint16_t	3055
adc_buf[22]	uint16_t	3060

Name : [2800...2899]

On passe en mode débuge pour voir le remplissage du buffer par la DMA.

main.c

Lib_Adrien.c

main.c

startup_stm321476rgtx.s

```

309  GPIO_InitStruct.Pull = GPIO_NOPULL;
310  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
311  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
312
313  /* USER CODE BEGIN MX_GPIO_Init_2 */
314  /* USER CODE END MX_GPIO_Init_2 */
315  }
316
317  /* USER CODE BEGIN 4 */
318  // Called when first half of buffer is filled
319  void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc)
320  {
321      // Set LD2
322      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
323  }
324
325  // Called when buffer is completely filled
326  void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
327  {
328      // Reset LD2
329      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
330  }
331  /* USER CODE END 4 */
332
333  /**
334   * @brief This function is executed in case of error occurrence.
335   * @retval None
336   */
337  void Error_Handler(void)
338  {
339      /* USER CODE BEGIN Error_Handler_Debug */
340      /* User can add his own implementation to report the HAL error return state */

```

Expr...

Regi...

Live ...

SFRs

Expression	Type	Value
[3900...3999]	uint16_t [10...	0x20001...
[4000...4095]	uint16_t [96]	0x20002...
adc_buf[4000]	uint16_t	3089
adc_buf[4001]	uint16_t	3074
adc_buf[4002]	uint16_t	3056
adc_buf[4003]	uint16_t	3048
adc_buf[4004]	uint16_t	3068
adc_buf[4005]	uint16_t	3068
adc_buf[4006]	uint16_t	3086
adc_buf[4007]	uint16_t	3069
adc_buf[4008]	uint16_t	3067
adc_buf[4009]	uint16_t	3072
adc_buf[4010]	uint16_t	3063
adc_buf[4011]	uint16_t	3057
adc_buf[4012]	uint16_t	3046
adc_buf[4013]	uint16_t	3072
adc_buf[4014]	uint16_t	3081
adc_buf[4015]	uint16_t	3087
adc_buf[4016]	uint16_t	3088
adc_buf[4017]	uint16_t	3042
adc_buf[4018]	uint16_t	3059
adc_buf[4019]	uint16_t	3073
adc_buf[4020]	uint16_t	3083
adc_buf[4021]	uint16_t	3045
adc_buf[4022]	uint16_t	3068

Name : [2800...2899]

Conclusion

Grâce à ces exercices, j'ai appris à utiliser :

- Le logiciel Logic
- ADC en mode polling
- ADC en mode DMA circulaire
- L'UART en TX pour avoir un retour sur un terminal
- L'UART en TX en mode DMA
- L'avantage de l'utilisation de la DMA sur un buffer volumineux
- L'impacte de la DMA sur le mode Débug