Universidad autónoma de baja California

Ingeniería en computación

Microcontroladores

Reporte practica 6: UART

Erik garcia Chávez 01275973

Jesús Adán Garcia López

28 de marzo del 2025

Teoría:

Manejo de UART:

el atmega 2560 cuenta con 4 UART's , el UART se compone de 3 "partes" tenemos el *clock generator, transmitter, receiver.* Los registros son compartidos.

El clock del UART se maneja bajo baudios, para poder saber la velocidad de este reloj es necesario una formula, tenemos 2 variantes uno que opera normal y otro a velocidad doble.

Equation for Calculating UBRR Value
$UBRRn = \frac{f_{OSC}}{16BAUD} - 1$
$UBRRn = \frac{f_{OSC}}{8BAUD} - 1$

Esta velocidad es con a que se estará comunicando con la terminal o incluso son otro UART's,

Esta velocidad se establece bajo el registro UBRR el cual es de 12 bits que cuenta con parte baja y parte alta.

El UART tiene 4 registros, el UBBR, tiene *UCSRnA, UCSRnB y UCSRnC,*

En el registro *UCSRnA* los bits que nos importa en esta práctica sería tan solo UDREn y U2X

UDREn : este bit indica que UDRn está vacío por lo que está listo para recibir nuevos datos

U2X: indica la velocidad, tenemos veleidad doble, con velocidad simple en la misma trama se mandan 16 muestras y con la velocidad doble se envían 8 muestras para la misma trama,

Registro UCSRnB, de igual forma los registros que por ahora nos interesan serán:

RXEn, TXEn : debemos habilitar estos bits para que el UART pueda funcionar como un puerto serial entonces ahora si puede recibir y transmitir, sus pines asociados pierden su propiedad digital y se vuelven un puerto serial

UCSZn2: en este registro, si se activa quiere decir que la trama de la información será de 9 bits.

Registro *UCSRnC* bits que nos interesa, este registro es el que tiene más bits para configurar nuestro UART.

UCSZn0, UCSZn1 : estos bits nos indican de que tamaño será la trama de la información, con esto 2 bits se puede configurar que se pueden enviar hasta 8 bits de información, si se quiere mandar 9 bits es necesario el bit extra en el registro *UCSRnB.*

UCSZn2	UCSZn1	UCSZn0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

USBSn: indica el stop bit, este le dice al UART que hasta aquí finaliza la trama de información, el cual pueden ser 1 bit o 2 bits

UPMn1, UPMn0: habilita el bit de paridad, el bit de paridad funciona para verificar posibles errores de trasmisión en la trama.

UMSEL1, UMSEL0: este registro selecciona qué modo de operación va a funcionar el UART, puede ser síncrono, asíncrono o master SPI, a nosotros para esta práctica manejaremos el modo asíncrono.

Secuencias de escape ASNI

las secuencias ASNI funcionan para especificar acciones como retornos de carro, movimiento de tabulación en terminales, también empleados para proporcionar representaciones literales de caracteres no imprimibles y de caracteres que normalmente tiene significados especiales.

estas son utilizadas principalmente en entorno de línea de comandos CLI para crear interfaces más interactivas o visualmente atractivas dado que se puede cambiar colores o posiciones dentro de la terminal

la estructura básica, esta secuencia comienza con el carácter **ESC (27 ASCII),** seguido de corchetes **[** parámetros que estos pueden tener son separados por **;** y una letra indica el comando, el carácter ESC puede ser represado como **\x1B[<parametors;P><comando>**

aquí tenemos unos ejemplos de estos:

Colors / Graphics Mode

ESC Code Sequence	Reset Sequence	Description
`ESC[1;34;{...}m`		Set graphics modes for cell, separated by semicolon (`;`).
`ESC[0m`		reset all modes (styles and colors)
`ESC[1m`	`ESC[22m`	set bold mode.
`ESC[2m`	`ESC[22m`	set dim/faint mode.
`ESC[3m`	`ESC[23m`	set italic mode.
`ESC[4m`	`ESC[24m`	set underline mode.
`ESC[5m`	`ESC[25m`	set blinking mode
`ESC[7m`	`ESC[27m`	set inverse/reverse mode
`ESC[8m`	`ESC[28m`	set hidden/invisible mode
`ESC[9m`	`ESC[29m`	set strikethrough mode.

EXPLICACION CODIGO:

```
1
2  #ifndef UART_H
3  #define UART_H
4
5  // estrucutra que define a los distintos uarts
6  typedef struct
7  {
8
9      volatile uint8_t UCSRA;
10     volatile uint8_t UCSRB;
11     volatile uint8_t UCSRC;
12     volatile uint8_t resb;
13     volatile uint16_t UBRR;
14     volatile uint8_t UDR;
15
16 } UART_reg_t;
17
18 /*
19 para poder acceder a ;ps registros
20
21 */
22
23 extern uint8_t *UART_offset[];
24
25 // Prototypes
26 // Initialization
27
28 UART_Ini(uint8_t com, uint32_t baudrate, uint8_t size, uint8_t parity, uint8_t stop);
29
30 uint16_t set_UBRR(uint32_t baudrate);
31
32 // Send
33 void UART_puts(uint8_t com, char *str);
34 void UART_putchar(uint8_t com, char data);
35
36 // Received*/
37 uint8_t UART_available(uint8_t com);
38 char UART_getchar(uint8_t com);
39 void UART_gets(uint8_t com, char *str);
40
41 // Escape sequences
42 UART_clrscr(uint8_t com);
43 UART_setColor(uint8_t com, uint8_t color);
44 UART_gotoxy(uint8_t com, uint8_t x, uint8_t y);
45
46 #define YELLOW 33 // Fixme
47 #define GREEN 32  // Fixme
48 #define BLUE 34   // Fixme
49
50 // Utils
51 void itoa(uint16_t number, char *str, uint8_t base);
52 uint16_t atoi(char *str);
53
54 void UART_putnum(uint8_t com, uint8_t num);
55
56 #endif
57
```

Como primeras instancias tenemos nuestro archivo cabecera en donde se están declarando los prototipos y variables globales que esteremos usando en el código, esta entre una directiva del preprocesador de C, esto se hace porque se está usando la cabecera en varios archivos para no evitar errores se declara una vez las variables y prototipos para que no ocurran errores durante la ejecución del código.

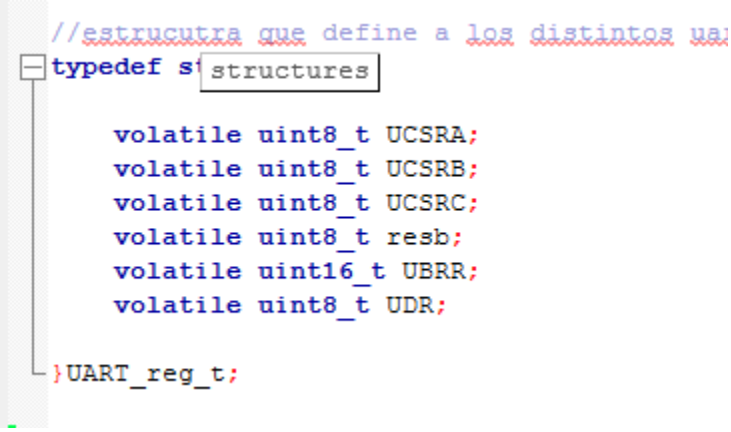
Para poder llamar a cualquier de los 4 UART's que tiene el atmega, se usa un arreglo por medio de apuntadores, estamos llamando a la dirección de memoria de x UART, dato que los registros de los UART entre si tienen la misma distribución de los registros, pero entre los espacios del UART 2 al 3, hay un gran espacio en memoria que se ocupa para algo más, para ser mucho más exacto y que el consigo sea muy reducido, usamos apuntadores, traemos el primer registro del UART

```
uint8_t *UART_offset[]=
{
    (uint8_t*)&UCSR0A,
    (uint8_t*)&UCSR1A,
    (uint8_t*)&UCSR2A,
    (uint8_t*)&UCSR3A

};
```

Con este registro dependiendo de lo que el usuario solicite iniciamos en el UART solicitado, UCSRnA es el registro inicial de cada UART. Podemos ver como esta distribuidos los registros en el atmega.

(0xC6)	UDR0	USART0 I/O Data Register								page 218
(0xC5)	UBRR0H	-	-	-	-	USART0 Baud Rate Register High Byte				page 222
(0xC4)	UBRR0L	USART0 Baud Rate Register Low Byte								page 222
(0xC3)	Reserved	-	-	-	-	-	-	-	-	
(0xC2)	UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0	page 235
(0xC1)	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80	page 234
(0xC0)	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0	page 234

Para poder usar estos registros hacemos uso de una estructura, la estructura tiene la misma disponibilidad en memoria que como en el atemga.



Con esto vamos a poder cualquier UART, esto va a depender del usuario.

UART_Ini:

la función que inicializa el UART debe ser muy genérica, lo que recibe como parámetros es que UART el usuario quiere usar, se le pasa el un valor al baud rate que se quiere llegar al final ese valor no es el que se le asigna a UBRR si no que es necesario pasar por una ecuación que el valor de la velocidad con las que el UART se va a comunicar.

Recibe el tamaño del frame que se va a transmitir, así como los bits de paridad que se va a querer y los stop bits, como estos son recibidos como 0, 1 o 2 en el caso del bit de paridad y 1 y 2 en el caso del stop bit hago usado de switch-case para los bits de paridad porque para 2 bits de paridad es 3 (0011) por lo que ocupamos cambiar lo nos manda a su representación adecuada para los bits en el registro lo mismo con los stop bits.

```
1
2  UART_Ini(uint8_t com, uint32_t baudrate, uint8_t size, uint8_t parity, uint8_t stop)
3  {
4
5      UART_reg_t *myUART = UART_offset[com]; // eligo a mi UART
6
7      myUART->UCSRB = (1 << TXEN0) | (1 << RXEN0); // Habilita TX y RX para UART0
8
9      uint8_t parity_mode = 0;
10
11     switch (parity)
12     {
13
14     case 1:
15         parity_mode = 3; // padidad impar
16         break;
17
18     case 2:
19
20         parity_mode = 2; // paridad par
21         break;
22
23     default:
24         parity_mode = 0; // en caso que no sea ninguno permanece en 0 pero por si
25         // alguna razon se mueve este nuermo lo ponemos de nuevo
26
27         break;
28     }
29
30     uint8_t stop_mode = (stop == 1) ? 0 : 1;
31
32     myUART->UCSRC = (parity_mode << UPM00) | (stop_mode << USBS0);
33
34     if (size == 9)
35     {
36         myUART->UCSRC |= (3 << UCSZ00); // UCSZ01:UCSZ00 = 0b11
37         myUART->UCSRB |= (1 << UCSZ02); // Habilitar bit 9
38     }
39     else
40     {
41         myUART->UCSRC |= ((size - 5) << UCSZ00); // Ej: 8 bits ? 3 << UCSZ00
42     }
43
44     uint16_t v_UBRR = (FOSC / (16 * baudrate)) - 1;
45     myUART->UBRR = v_UBRR;
46 }
```

SEND:

UART_PUTCHAR

La función del putchar es la de enviar un único carácter (byte) a través del módulo UART. Es una función bloqueante lo que quiere decir es que espera hasta que el módulo UART esté listo para transmitir antes de enviar el dato.

Como tuve problemas usando UDRE de manera genérica, mediante unas condiciones traemos el que le corresponde a cada UART. El UDRE es el bit de UCSRA que indica si el buffer de transmisión esta vacío.

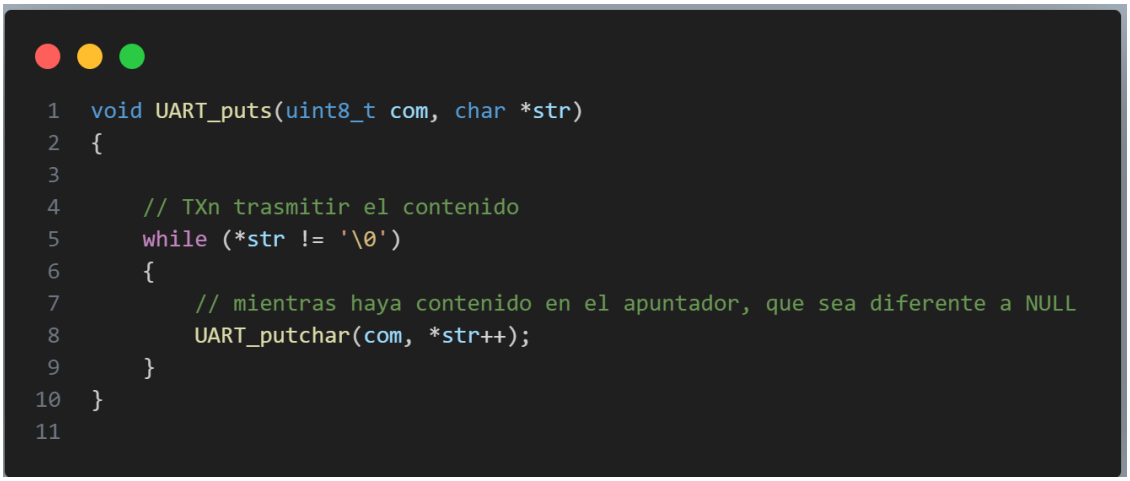
Aplicamos una máscara a UDRE porque solo nos interesa ese bit, el registro UDRE está en 1 cuando es buffer esta vacío, por lo que al aplicar una negación estamos diciendo que si está lleno cunado está en 0 este va a esperar hasta que se vacié, y cunado este está vacío (1) pero con la negación 0 este empieza a trasmitir el contenido en el registro UDR

Lo que está recibiendo es el puerto por el cual se va enviar el frame así como el dato que se quiere enviar

```c
void UART_putchar(uint8_t com, char data)
{

    UART_reg_t *myUART = UART_offset[com];

    // Calcular el bit UDRE según el UART (ej: UDRE0 para com=0, UDRE1 para com=1)
    uint8_t udre_bit = (com == 0) ? UDRE0 : (com == 1) ? UDRE1
                                        : (com == 2)   ? UDRE2
                                                       : UDRE3;

    // va a esperar hasta que se vacie por completo
    while (!(myUART->UCSRA & (1 << udre_bit)))
        ;
    ; // espera a que el periferico este vacio

    myUART->UDR = data;
}

```

UART_ puts:

La función lo único que está haciendo es mandar a llamar a la función *UART_PUTCHAR* este lo manda a llamar hasta que en el string encuentre el carácter vacío lo que indica que el carácter llego hasta su máxima capacidad y es el final, como es un apuntador realmente no sabemos que de tamaño es por lo que agrego el carácter nulo ‘\0’ al final de la cadena para indicar ese final.



```
void UART_puts(uint8_t com, char *str)
{

    // TXn trasmitir el contenido
    while (*str != '\0')
    {
        // mientras haya contenido en el apuntador, que sea diferente a NULL
        UART_putchar(com, *str++);
    }
}

```

RECIVED

UART_getchar():

Esta función se encarga de recibir un carácter (byte) desde un módulo UART especifico por esa razón solo recibe como parámetro el puerto del cual va a leer

Mediante la estructura y el arreglo seleccionamos el UART que vamos a usar.

Esta función espera hasta que reciba un dato, haciendo una máscara al bit RXCn (USART RECEIVE COMPLET) el cual está en alto cunado hay un dato sin leer dentro del buffer y está en 0 cuando esta vacío. Por lo que al negar esta condición le estamos indicando que espere mientras RXC no tenga datos disponibles, cunado RX reciba un dado la condición será 0 por lo que no entra al bucle y pasa a leer UDR retornar el valor.

```
uint8_t UART_available(uint8_t com)
{

    // RXC0 sta en 1 cuando hay un dato sin leer en RXC
    // y esta en 0 cunado este no tiene nada

    UART_reg_t *myUART = UART_offset[com];

    return (myUART->UCSRA & (1 << RXC0)); // Hay dato disponible
                                          // creo que va a asi pero si hay errores podemos invertirlo
}

char UART_getchar(uint8_t com)
{
    UART_reg_t *myUART = UART_offset[com];
    while (!(myUART->UCSRA & (1 << RXC0)))
        ; // Espera dato
    return myUART->UDR;
}

```

UART_gets():

Esta función tiene la función de recopilar todos los caracteres hasta un tamaño especifico así como fungir como árbitro dependiendo de los diretentes sucesos que pueden ingresar por la terminal. Tenemos varias partes

En primero tenemos un super loop, esto porque no sabemos hasta cuanto es el buffer que tiene en este momento, tenemos varias variables que funcionan de la siguiente manera, **C** funciona para capturar el dato que se recibió pro *getchar,* **i** funciona como el índice que indica en que posición y que tamaño tiene en este momento el apuntador del string capturado por el UART, **dot_flat** es una variable especial que uso en el caso que se introduzcan puntos.

Lo primero que se hace es capturar el carácter que se recibió *UART_getchar,* en el caso que el usuario decida borrar un carácter es la primera condición que tenemos, esto porque, porque si el usuario borra algo es lo primero que debe de pasar antes de

mostrar, el carácter **‘\b’** me indica que se está presionando la tecla *backspace* pero tenemos una condición en el caso que **i>0** no queremos que borre lo que está en la terminal en este caso sería una instrucción, porque eso no forma parte de lo que el usuario ha introducido por lo que lo que hacemos es pasar de largo y continuar con el programa pero en el caso de que el usuario haya ingresado algún elemento, va a retroceder el índice dado que se está liberando ese espacio **str[--i] = ‘\0’** primero se retrocede y después se remplaza el ultimo carácter con el carácter nulo

Con:

UART_putchar(com, '\b'); Mueve el cursor a la izquierda.

UART_putchar(com, ' '); Sobrescribe el carácter con un espacio.

UART_putchar(com, '\b'); Mueve el cursor a la izquierda nuevamente.

Saliendo de la condición mostramos por pantalla lo que tiene **C.**

```
1
2  char c; // este va a capturar el valor del char que se introdujo
3  uint8_t i = 0;
4  uint8_t dot_flag = 0; // Bandera para detectar punto
5
6  while (1)
7  {
8      c = UART_getchar(com);
9      if (c == '\b')
10     {
11         if (i > 0)
12         {
13
14             str[--i] = '\0'; // sustitumos el utlimo caracter con el nulo
15             UART_putchar(com, '\b');
16             UART_putchar(com, ' ');
17             UART_putchar(com, '\b');
18         }
19
20         continue; // si no hay nada que borrar o si hay algoq ue borrar sigue con el ciclo
21     }
22
23     UART_putchar(com, c);
24
```

Como saber que la cadena ya termino, una parte claro el carácter vacío ‘\0’ pero como saber que la función gets ya dejarla, porque como se ve es un ciclo infinito, por lo que la forma de saber que ya no queremos ingresar nada mas es con el carácter ‘\n’ que sería **enter** en nuestro teclado. Por lo que estamos constantemente buscando lo que esta introduciendo por UART, pero si es el carácter de salto de línea quiere decir que el usuario ya termino de ingresar los datos, por lo que finalizamos la cadena con el carácter vacío en la posición de **i** así que hacer un retorno de carro para ir al inicio de la terminal y brincamos a la línea siguiente.

```
1
2  if(c == '\r' || c== '\n'){
3      str[i]='\0'; //caracter nulo denotando que la
4      UART_putchar(com,'\r'); //vuelve al inicio de la linea
5      UART_putchar(com,'\n'); //salto de linea
6      break; //rompesmos el ciclo y a esperar que se vuelva a escribir algo
7  }
8
```

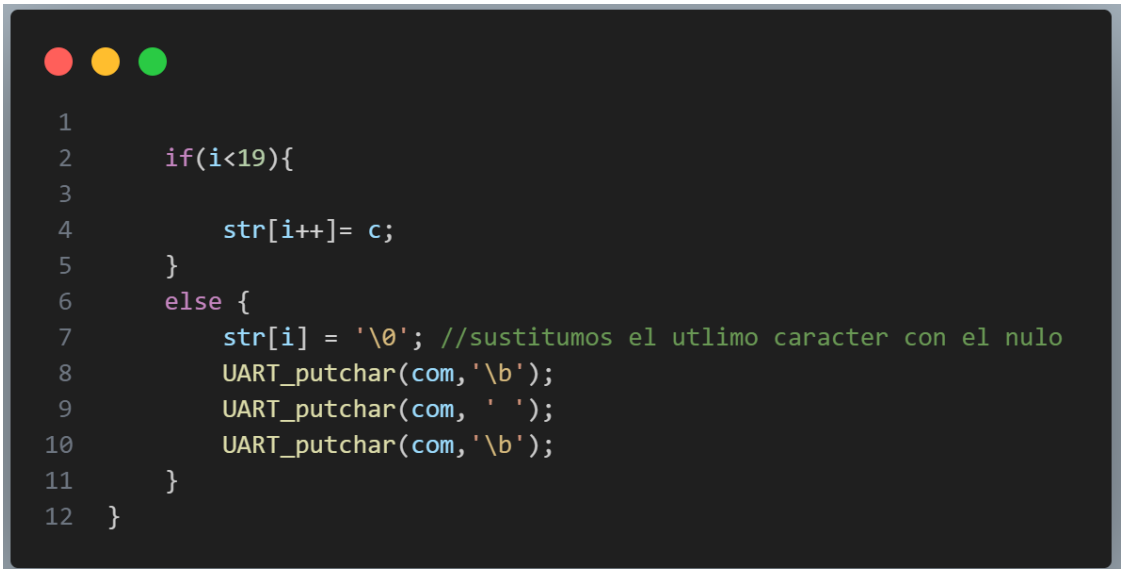
Caso especial de (.)

Evitamos procesar número con punto decimal para evitar posibles errores, por lo que la idea es que si mira un punto entre la información ingresada tengo una bandera que indica si se ingresó un punto, en la condición donde entra cunado la bandera esta activada ponemos el carácter nulo para terminar donde está el punto, esto porque sabemos que el puts imprime hasta que encuentre el carácter nulo, esto se hace para que lo que este después del punto lo ignore, se imprime por UART pero está haciendo ignorado para ser procesado con **itoa** y **atoi,** después la borramos para que ya no entre no necesitamos más porque con el único donde está el punto es más que suficiente.

```
1
2  if (c == '.') {
3      dot_flag = 1;
4      continue; // No muestra el punto
5  }
6
7  if(dot_flag){
8      str[i++] = '\0'; //desoues de este punto ya no lo tomara en cuenta
9      dot_flag = 0; //para que ya no enre aqui
10 }
11
```

Límite de introducir.

Ponemos un límite de los datos que se muestran por UART, verificamos que el índice sea menor a la cantidad limite, esto porque el ultimo índice estaría reservado para el carácter nulo ‘\n’, para que no siga mostrando lo que se hace es algo parecido al borrado, constantemente estamos remplazando ese nuevo carácter con un espacio vacío y retrocedemos cursor constantemente



```
1
2       if(i<19){
3
4           str[i++]= c;
5       }
6       else {
7           str[i] = '\0'; //sustitumos el utlimo caracter con el nulo
8           UART_putchar(com,'\b');
9           UART_putchar(com, ' ');
10          UART_putchar(com,'\b');
11      }
12  }
```

```
1
2  void UART_gets(uint8_t com, char *str)
3  {
4
5      char c; // este va a capturar el valor del char que se introdujo
6      uint8_t i = 0;
7      uint8_t dot_flag = 0; // Bandera para detectar punto
8
9      while (1)
10     {
11         c = UART_getchar(com);
12         if (c == '\b')
13         {
14             if (i > 0)
15             {
16
17                 str[--i] = '\0'; // sustitumos el utlimo caracter con el nulo
18                 UART_putchar(com, '\b');
19                 UART_putchar(com, ' ');
20                 UART_putchar(com, '\b');
21             }
22
23             continue; // si no hay nada que borrar o si hay algoq ue borrar sigue con el ciclo
24         }
25
26         UART_putchar(com, c);
27
28         if (c == '\r' || c == '\n')
29         {
30             // retorno de carro o salto de linea lo que quiere decir que se terminao de escribir el
31             // texto actual.
32
33             str[i] = '\0';           // caracter nulo denotando que la
34             UART_putchar(com, '\r'); // vuelve al inicio de la linea
35             UART_putchar(com, '\n'); // salto de linea
36             break;                   // rompesmos el ciclo y a esperar que se vuelva a escribir algo
37         }
38
39         if (c == '.')
40         {
41             dot_flag = 1;
42             continue; // No muestra el punto
43         }
44
45         if (dot_flag)
46         {
47
48             str[i++] = '\0'; // desoues de este punto ya no lo tomara en cuenta
49             dot_flag = 0;    // para que ya no enre aqui
50         }
51
52         // para 20 caracteres, si no lo regresamos a 127
53
54         if (i < 19)
55         {
56
57             str[i++] = c;
58             // UART_putchar(com, c);
59         }
60         else
61         {
62             str[i] = '\0'; // sustitumos el utlimo caracter con el nulo
63             UART_putchar(com, '\b');
64             UART_putchar(com, ' ');
65             UART_putchar(com, '\b');
66         }
67     }
68 }
```

Funciones de escape:

Como vimos en la teoría, las secuencias de escape describen propiedades y acciones que debe de realizar la terminal, podemos encontrar las diferentes instrucciones si se busca por internet como en la siguiente liga:
https://www2.ccs.neu.edu/research/gpc/VonaUtils/vona/terminal/vtansi.htm

UART_clrscr():

Esta secuencia lo que hace es limpiar la pantalla.

```
UART_clrscr(uint8_t com)
{

    UART_reg_t *myUART = UART_offset[com];
    UART_puts(com, "\x1B[2J"); // borra toda la pantalla
    UART_puts(com, "\x1B[H");  // poen el curso al incio fila 1, columna 1
}
```

UART_putnum() – UART_gotoxy()

La función putnum es convertir un numero en sus dígitos ASCII y los envía carácter por carácter. Las condiciones extraen las centenas y decenas la operación adentro convertimos el numero en su versión ASCII.

La función gotoxy mueve el cursor a una posición especifica en la pantalla, la secuencia de escape es **\x1B[y;xH**

```
1
2  void UART_putnum(uint8_t com, uint8_t num)
3  {
4
5      if (num >= 100)
6      {
7          UART_putchar(com, '0' + (num / 100));
8          num %= 100;
9      }
10     if (num >= 10)
11     {
12         UART_putchar(com, '0' + (num / 10));
13         num %= 10;
14     }
15
16     UART_putchar(com, '0' + num);
17 }
18
19 UART_gotoxy(uint8_t com, uint8_t x, uint8_t y)
20 {
21
22     UART_puts(com, "\x1B["); // inicio de la secuencia de esapce
23
24     UART_putnum(com, y + 1); // convertir a caracter
25     UART_putchar(com, ';');
26     UART_putnum(com, x + 1);
27     UART_putchar(com, 'H'); // final de la secuencia
28 }
```

UART_setcolor():

Para establecer el color se hace mediante la secuencia de escape **\x1B[<codigo>m**

Donde código es el número del color, donde tenemos los siguientes

Color Name	Foreground Color Code	Background Color Code
Black	30	40
Red	31	41
Green	32	42
Yellow	33	43
Blue	34	44
Magenta	35	45
Cyan	36	46
White	37	47
Default	39	49

Como los pasados operamos con enviarlos por separado el comando.

Lo que se hace con putchar en enviar las decenas y unidades para enviar cada digito como carácter ASCII

```
1
2  UART_setColor(uint8_t com, uint8_t color)
3  {
4
5      UART_puts(com, "\x1B["); // incio del comando espace
6      UART_putchar(com, '0' + (color / 10));
7      UART_putchar(com, '0' + (color % 10));
8      UART_putchar(com, 'm'); // final del comando
9  }
```

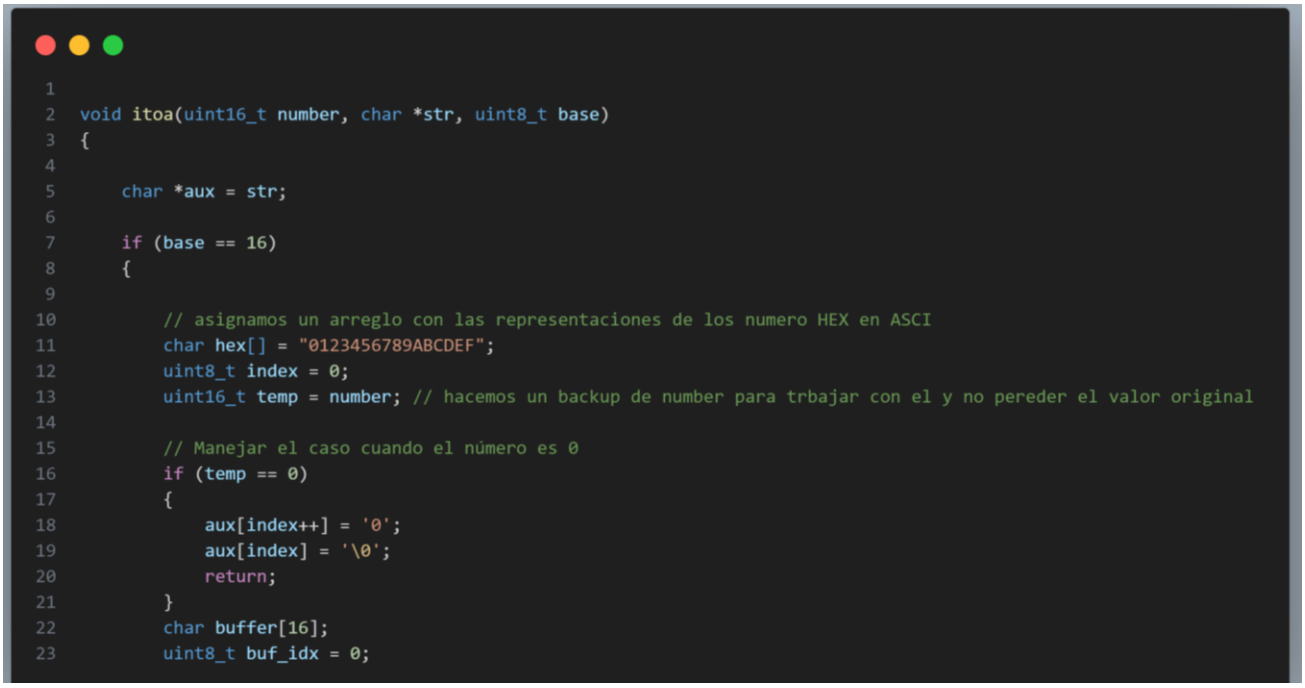
Itoa:

Itoa convierte un valor numérico a su representación HEX, pero en ASCII para poder ser mostrado por UART. La función convierte tanto binario como HEX por lo que estará recibiendo el numero que anteriormente se convirtió en atoi, el apuntador de la cadena, así como la base a la cual se quiere convertir, se utiliza una variable auxiliar para no perder el puntero inicial de la cadena.

Hacemos una matriz la cual tiene las letras que representa los números en HEX, así como una variable que tiene la cantidad igual que **number** esto para que el valor original no se pierda.

Primero verificamos que el numero no sea cero, si es así no tiene caso el que se haga un proceso que solo seria tiempo perdido.

Asignamos un arreglo **buffer** que es donde se estará guardando la representaciones HEX en ASCII del número que estamos convirtiendo así como un contador que funcionaria como que tamaño es el digito.



```
1
2  void itoa(uint16_t number, char *str, uint8_t base)
3  {
4
5      char *aux = str;
6
7      if (base == 16)
8      {
9
10         // asignamos un arreglo con las representaciones de los numero HEX en ASCI
11         char hex[] = "0123456789ABCDEF";
12         uint8_t index = 0;
13         uint16_t temp = number; // hacemos un backup de number para trbajar con el y no pereder el valor original
14
15         // Manejar el caso cuando el número es 0
16         if (temp == 0)
17         {
18             aux[index++] = '0';
19             aux[index] = '\0';
20             return;
21         }
22         char buffer[16];
23         uint8_t buf_idx = 0;
```

La operación correcta para calcular la representación HEX de un numero decimal es mediante la división modular, es decir el resto de la división, el resto de esa división es la representación del número decimal, se divide dependiendo la base, para HEX entre 16, el numero resultante es exactamente el índice en nuestro arreglo **hex[]** es decir que el resultado de la división modular es 11 el índice 11 de nuestro arreglo es el **B** por lo que se selecciona ese elemento, cada elemento se va agregando al segundo arreglo el llamado **buffer** el cual funciona para almacenar cada carácter, vamos reduciendo el número haciendo una división entera sobre la misma base

Como sacamos los valores ahora?, con este método estamos haciendo como una pila, esto porque el primer elemento que se calcula es el último elemento en la representación del numero HEX, es decir si se calcula **[B,0,2]** que sería la representación HEX de 523 pero **B02** realmente no es **523** el número esta al revés, por lo que es como una pila primero se saca el ultimo elemento en entrar a esa pila, para eso hacemos uso de un ciclo for dado que conocemos la longitud del número que del tamaño de **buf_idx** como la cadena es un apuntador empezamos a remplazar desde el inicio y al final volvemos a agregar el carácter nulo denotando el final de dicha cadena.

```
1
2  while (temp > 0)
3  {
4      buffer[buf_idx++] = hex[temp % base];
5      temp /= base;
6  }
7
8  for (int i = buf_idx - 1; i >= 0; i--)
9  {
10     aux[index++] = buffer[i];
11 }
12 aux[index] = '\0';
```

Para cuando es binario es algo parecido, al inicio tenemos nuestra condición en caos que sea cero, así como nuestras variables locales que son copias de los argumentos, así común arreglo que funciona la **buffer** en donde se guarda el numero en su representación binaria.

```
1
2  else if (base == 2)
3  {
4
5      uint8_t index = 0;
6      uint16_t temp = number;
7
8      char buffer[17]; // tiene tamaño 16 porque el number es un numero de 16 bits, el caracter nulo
9      // se agrega despues en el apuntador
10
11     uint8_t buf_idx = 0;
12     if (number == 0)
13     {
14
15         aux[index++] = '0';
16         aux[index] = '\0';
17         return;
18     }
```

La manera de calcular si es 1 o si es 0, es muy parecido al de HEX usando la división modular, pero esta es diferente, si el resto es 0 es obvio que el bit es ‘0’ pero si es diferente, como 5 entonces el bit serio ‘1’, por lo que hacemos lo mismos de agregar cada bit al buffer, e igual que con HEX el ultimo bit en entrar es el MSB y el primero en entrar es el LSB.

Agregamos ceros una posición después de en donde se quedó el índice el buffer, porque no se sabe que puede haber después muy posiblemente haya basura y no queremos eso.

```
    while (temp > 0)
    {
        buffer[buf_idx++] = (temp % 2) ? '1' : '0';
        temp /= 2;
    }

    // rellenar con ceros a la izqueirda

    while (buf_idx < 16)
    {
        buffer[buf_idx++] = '0';
    }

    index = 0;

    for (int8_t i = buf_idx - 1; i >= 0; i--)
    {
        aux[index++] = buffer[i];
    }

    aux[index] = '\0';
}
```

```c
void itoa(uint16_t number, char* str, uint8_t base){

    char *aux= str;

    if(base == 16){

        //asignamos un arreglo con las representaciones de los numero HEX en ASCI
        char hex[] = "0123456789ABCDEF";
        uint8_t index = 0;
        uint16_t temp = number; //hacemos un backup de number para trbajar con el y no pereder el valor original

        // Manejar el caso cuando el número es 0
        if (temp == 0) {
            aux[index++] = '0';
            aux[index] = '\0';
            return;
        }
        char buffer[16];
        uint8_t buf_idx = 0;

        while (temp > 0) {
            buffer[buf_idx++] = hex[temp % base];
            temp /= base;
        }

        for (int i = buf_idx - 1; i >= 0; i--) {
            aux[index++] = buffer[i];
        }
        aux[index] = '\0'; // Terminar con nulo
    }

    else if(base == 2){

        uint8_t index = 0;
        uint16_t temp = number;

        char buffer[17]; //tiene tamaño 16 porque el number es un numero de 16 bits, el caracter nulo
        //se agrega despues en el apuntador

        uint8_t buf_idx=0;
        if(number ==0){

            aux[index++] = '0';
            aux[index] = '\0';
            return;
        }

        while(temp > 0){

            buffer[buf_idx++]= (temp%2)?'1':'0';
            temp/=2;
        }

        //rellenar con ceros a la izqueirda

        while(buf_idx < 16){
            buffer[buf_idx++]= '0';
        }

        index=0;

        for(int8_t i = buf_idx-1 ; i >= 0 ; i--){
            aux[index++] = buffer[i];
        }

        aux[index]='\0';

    }

}
```

Atoi:

Esta función lo que hace es recoge los valores numéricos en ASCII y los convierte en su valor numérico. Hacemos uso de una variable que funge como acumulador

Lo que se hace es recorrer el string hasta llegar al final **'\0'** mientras que este entre el valor numérico decimal en carácter

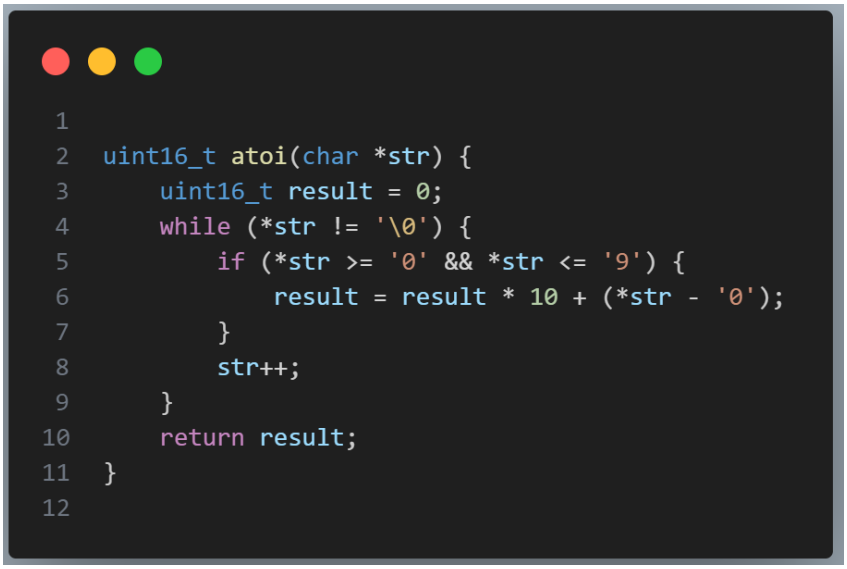
La operación que se hace es primero calcular en su visión numérica que numero es el que se está tratando, por ejemplo si tenemos '1' en su versión ascii seria 49, a todos los números si le restamos 48 -> '0' el resultado sería su valor pero en versión numérica, 1, 2, 3, etc. Pero como saber si el digito que sacamos de esa resta corresponde a una decena, una centena, etc. Lo que se hace es que al resultado pasado lo multiplicamos por 10, esto para calcular su verdadero valor, y todo se va sumando por ejemplo si tenemos '123'

Calcula:

0 * 10 + 1 = 1

1 * 10 + 2 = 12

12 *10 + 3 = 123



```
1
2  uint16_t atoi(char *str) {
3      uint16_t result = 0;
4      while (*str != '\0') {
5          if (*str >= '0' && *str <= '9') {
6              result = result * 10 + (*str - '0');
7          }
8          str++;
9      }
10     return result;
11 }
12
```

Conclusiones:

En la practica hacemos uso del periférico de comunicación UART, es un periférico se comunicación serial, muy importante y esencial para comunicarse entre micros o con alguna terminal.

Alguna de las partes que mas se me complico, puede el cómo ingresar datos o mandar datos por este, pero una vez que entendí la teoría y como funcionan los registros comprendí como funciona, así como la función gets, el cómo interpretar las distintas opciones que el usuario puede ingresar, incluso no contemple si se introduce un punto decimal cortar el número, eso se hizo después pero al final no fueron cosas muy complejas más allá de comprender como es que trabaja el periférico.