Университет ИТМО

Кафедра Вычислительной Техники

Лабораторная работа №2

по дисциплине “Встроенные системы”

Группа P3401

Комаров Егор Андреевич

Вариант 1

Оценка:_____

Принял: Ключев Аркадий Олегович

Санкт-Петербург, 2020

Задание

Драйвер UART должен каждый принимаемый контроллером символ отсылать назад – так называемое «эхо». Каждое новое сообщение от стенда должно выводиться с новой строки.

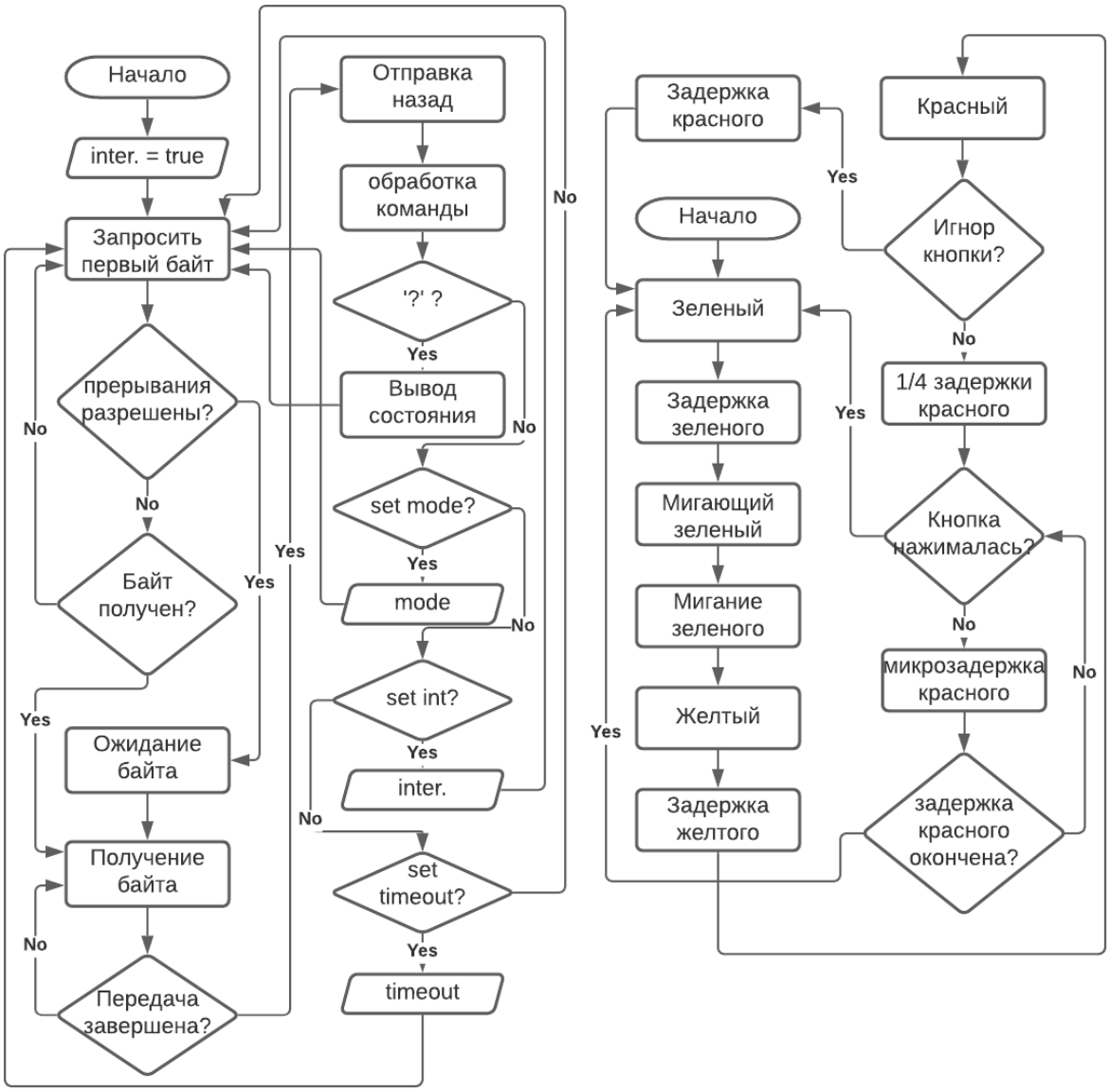
Связь между стендом и персональным компьютером (ПК) осуществляется с использованием терминальной программы – например, putty. Физически связь между ПК и стендом осуществляется через тот же кабель USB, через который происходит прошивка и отладка – со стороны SDK находится микросхема, которая управляется с помощью USB и формирует последовательные сигналы UART, которые подаются на входы контроллера. Реализуется два варианта программы: с использованием прерываний и без.

Доработать программу «светофор», добавив возможность отключения кнопки и задание величины таймаута (период, в течение которого горит красный).

Должны обрабатываться следующие команды, посылаемые через UART:

- *?* – в ответ контроллер должен прислать состояние, которое отображается в данный момент на диодах: *red*, *yellow*, *green*, *blinking green*, режим – *mode* 1 или *mode 2*, величину таймаута (сколько горит красный) – *timeout …*, и задействованы ли прерывания или нет – символ *I* или *P*;
- *set mode* 1 или *set mode 2* – установить режим работы светофора, когда обрабатываются или игнорируются нажатия кнопки;
- *set timeout X* – установить таймаут;
- *set interrupts on* или *set interrupts off* – включить или выключить прерывания в драйвере.

Блок-схема



Инструментарий

Аппаратные средства

Uart

В ходе работы инициализируется 2 интерфейса UART с целью подключения друг к другу, однако, из-за невозможности осуществить такую связь удаленно используется только один из них:

- UART2 на портах GPIO PA2 (TX) и PA3 (RX)
- UART3 на портах GPIO PB10 (TX) и PB11 (RX)

Характеристики:

- Асинхронный режим

- 115200 bps
- 8 бит сообщение
- Нет бита четности
- Стоп-бит – 1
- Receive и Transmit

Программные средства

Асинхронная передача через UART

Для асинхронной передачи через UART используются **HAL_UART_Receive_IT** и **HAL_UART_Transmit_IT**. При этом выполнение не блокируется, а при завершении операции возможно прерывание

Синхронная передача через UART

Для синхронной передачи через UART используются **HAL_UART_Receive** и **HAL_UART_Transmit**. При этом выполнение блокируется до тех пор, пока не будет передано/принято указанное количество байт либо истечет timeout

Использование прерываний UART

Для обработки прерываний использован callback **HAL_UART_RxCpltCallback** при завершении операции на прием (и **HAL_UART_TxCpltCallback** на передачу), переопределенный с пользовательским кодом

Инициализация UART

Для инициализации использовалась функция **HAL_UART_Init,** принимающая структуру с параметрами UART (скорость, длина слова, параметры стоп битов, бит четности, режим чтения/записи и др)

Подключение UART к GPIO

Для подключения UART к устройствам используются GPIO порты в режиме **GPIO_MODE_AF_PP** и заданной альтернативной периферией в виде нужно UART

Исходный код

SDK

В качестве основы была взят код из прошлой лабораторной работы.

Uart

В качестве API были добавлены функции для управления прерываниями и передачи данных известного размера. Приёмом управляет драйвер, который отсылает принятые данные обработчику команд.

SDK/interface.h

```
// uart
#define SDK_UART_HANDLE huart3
#define SDK_UART_TIMEOUT 3
#define SDK_UART_BUFFER_SIZE 128

// uart API
void SDK_UART_Init();
void SDK_UART_EnableInterrupts(bool interrupts);
void SDK_UART_Transmit(uint8_t* pData, size_t size);
bool SDK_UART_IsInterruptible();
```

Прием данных начинается с инициализации SDK_UART. Сначала происходит ожидание получения первого байта в начало внутреннего буфера и затем побайтово записываются данные, пока не кончатся. После этого синхронно обрабатывается полученная команда и начинается новая итерация считывания.

SDK/uart.c

```
/// TYPES ///
typedef struct UartData
{
	bool m_interrupts;
	uint8_t m_buffer[SDK_UART_BUFFER_SIZE];
} UartData;

/// STATIC ///
static UartData s_uartData;
static void SDK_UART_StartReceiving();
static void SDK_UART_ContinueReceiving();
static void SDK_UART_Receive(uint8_t* pData, size_t size, size_t offset);

static void SDK_UART_StartReceiving()
{
	if(s_uartData.m_interrupts)
	{
		// will be recieved by callback
		HAL_UART_Receive_IT(&SDK_UART_HANDLE, s_uartData.m_buffer, 1);
	}
	else
	{
		// wait for first byte
		while(HAL_UART_Receive(&SDK_UART_HANDLE, s_uartData.m_buffer, 1,
SDK_UART_TIMEOUT) != HAL_OK) { }
		SDK_UART_ContinueReceiving();	}
}
static void SDK_UART_ContinueReceiving()
{
	SDK_UART_Receive(s_uartData.m_buffer, SDK_UART_BUFFER_SIZE, 1 );
	SDK_MAIN_ProcessCommand((const char*)s_uartData.m_buffer);
	SDK_UART_StartReceiving();
}

/// API ///
void SDK_UART_Init()
{
	s_uartData.m_interrupts = true;
	SDK_UART_StartReceiving();
}
void SDK_UART_EnableInterrupts(bool interrupts)
{
	s_uartData.m_interrupts = interrupts;
```

```
}

void SDK_UART_Transmit(uint8_t* pData, size_t size)
{
	HAL_UART_Transmit(&SDK_UART_HANDLE, pData, size, SDK_UART_TIMEOUT);

	SDK_DBG_Print("Uart: %s", pData);
}

void SDK_UART_Receive(uint8_t* pData, size_t size, size_t offset)
{
	uint32_t dataInd = offset;

	while(dataInd < size - 1 &&
			HAL_UART_Receive(&SDK_UART_HANDLE, pData + dataInd, 1,
SDK_UART_TIMEOUT) == HAL_OK)
	{
		dataInd++;
	}
	pData[dataInd] = 0;
	SDK_UART_Transmit(pData, dataInd);
}


bool SDK_UART_IsInterruptible()
{
	return s_uartData.m_interrupts;
}
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
	SDK_UART_ContinueReceiving();
}
```

Timer

Для имитации приема команд аналогично нажатию кнопки из первой лабы были добавлены отложенные вызовы, но уже несколько штук и с аргументом. В остальном различий нет.

SDK/interface.h

```
// timer
#define SDK_TIM_INTERRUPT_MAX 5
#define SDK_TIM_DATA_INTERRUPT_MAX 15

// timer API
void SDK_TIM_Update();
void SDK_TIM_AddInterrupt(void(*callbackPtr)(), uint32_t delay, bool periodic);
void SDK_TIM_AddDataInterrupt(void(*callbackPtr)(void* data), void* data, uint32_t delay,
bool periodic);
```

SDK/timer.c

```
typedef struct Interrupt
{
	void (*m_callbackPtr)();
	uint32_t m_delay;
	uint32_t m_timer;
	bool m_periodic;
} Interrupt;
typedef struct DataInterrupt
{
	void (*m_callbackPtr)(void* data);
```

```
	void* m_data;
	uint32_t m_delay;
	uint32_t m_timer;
	bool m_periodic;
} DataInterrupt;

/// STATIC ///
static Interrupt s_interrupt[SDK_TIM_INTERRUPT_MAX];
static uint32_t s_interruptCount = 0;

static DataInterrupt s_dataInterrupt[SDK_TIM_DATA_INTERRUPT_MAX];
static uint32_t s_dataInterruptCount = 0;


/// API ///
void SDK_TIM_AddInterrupt(void(*callbackPtr)(), uint32_t delay, bool periodic)
{
	if(s_interruptCount < SDK_TIM_INTERRUPT_MAX)
	{
		s_interrupt[s_interruptCount].m_callbackPtr = callbackPtr;
		s_interrupt[s_interruptCount].m_delay = delay;
		s_interrupt[s_interruptCount].m_timer = 0;
		s_interrupt[s_interruptCount].m_periodic = periodic;

		s_interruptCount++;
	}
}
void SDK_TIM_AddDataInterrupt(void(*callbackPtr)(void* data), void* data, uint32_t delay,
bool periodic)
{
	if(s_dataInterruptCount < SDK_TIM_DATA_INTERRUPT_MAX)
	{
		s_dataInterrupt[s_dataInterruptCount].m_callbackPtr = callbackPtr;
		s_dataInterrupt[s_dataInterruptCount].m_data = data;
		s_dataInterrupt[s_dataInterruptCount].m_delay = delay;
		s_dataInterrupt[s_dataInterruptCount].m_timer = 0;
		s_dataInterrupt[s_dataInterruptCount].m_periodic = periodic;

		s_dataInterruptCount++;
	}
}
void SDK_TIM_Update()
{
	for(int i = 0; i < s_interruptCount; ++i)
	{
		if(s_interrupt[i].m_callbackPtr && ++s_interrupt[i].m_timer >=
s_interrupt[i].m_delay)
		{
			s_interrupt[i].m_timer = 0;
			s_interrupt[i].m_callbackPtr();
			if(!s_interrupt[i].m_periodic)
			{
				s_interrupt[i].m_callbackPtr = 0;
			}
		}
	}

	for(int i = 0; i < s_dataInterruptCount; ++i)
	{
		if(s_dataInterrupt[i].m_callbackPtr && ++s_dataInterrupt[i].m_timer >=
s_dataInterrupt[i].m_delay)
		{
```

```
                    s_dataInterrupt[i].m_timer = 0;
                    s_dataInterrupt[i].m_callbackPtr(s_dataInterrupt[i].m_data);
                    if(!s_dataInterrupt[i].m_periodic)
                    {
                          s_dataInterrupt[i].m_callbackPtr = 0;
                    }
              }
       }
}
```

Main

В главную обертку была добавлена функция для обработки данных, принятых из UART. Также требует определения в контексте приложения.

SDK/interface.h

```
void SDK_MAIN_ProcessCommand(const char* command);
```

Application

Semaphore

К светофору было добавлено несколько новых свойств согласно заданию.

App/semaphore.h

```
/// TYPES ///
typedef enum eColorState
{
       ECS_Green,
       ECS_Yellow,
       ECS_Red,
       ECS_BlinkingGreen
} eColorState;

typedef enum eSemaphoreMode
{
       ESM_ProcessPress = 1,  // mode1
       ESM_IgnorePress = 2    // mode2
} eSemaphoreMode;

typedef struct SemaphoreState
{
       eColorState m_color;
       eSemaphoreMode m_mode;
       uint32_t m_redTimeout;
} SemaphoreState;

SemaphoreState SEM_GetState();
void SEM_SetMode(eSemaphoreMode mode);
void SEM_SetRedTimeout(uint32_t timeout);
```

App/semaphore.c

```
/// STATIC ///
static SemaphoreState s_semaphoreState;

static uint16_t MapColorStateToLed(eColorState color)
{
       switch(color)
       {
```

```
        case ECS_Green:
                return SDK_LED_GREEN;
        case ECS_Yellow:
                return SDK_LED_YELLOW;
        case ECS_Red:
                return SDK_LED_RED;
        case ECS_BlinkingGreen:
                return SDK_LED_GREEN;
        }
        return 0;
}
static eColorState MapLedToColorState(uint16_t led, bool isBlinking)
{
        switch(led)
        {
        case SDK_LED_GREEN:
                return isBlinking ? ECS_BlinkingGreen : ECS_Green;
        case SDK_LED_YELLOW:
                return ECS_Yellow;
        case SDK_LED_RED:
                return ECS_Red;
        }
        return 0;
}

/// API ///
void SEM_Init()
{
        s_semaphoreState.m_color = ECS_Red;
        s_semaphoreState.m_mode = ESM_ProcessPress;
        s_semaphoreState.m_redTimeout = SEM_RED_PERIOD;
}

.....................
        // red
        SEM_Show(SDK_LED_RED, s_semaphoreState.m_redTimeout, s_semaphoreState.m_mode ==
ESM_ProcessPress);
.....................

SemaphoreState SEM_GetState()
{
        return s_semaphoreState;
}
void SEM_SetMode(eSemaphoreMode mode)
{
        s_semaphoreState.m_mode = mode;
}
void SEM_SetRedTimeout(uint32_t timeout)
{
        s_semaphoreState.m_redTimeout = timeout;
}
```

App cycle

В цикле приложения при старте откладывается несколько тестовых команд.

App/application.c

```
void SDK_MAIN_PreLoop()
{
        SDK_DBG_Print("%s", "Begin simulation");
        // init semaphore
```

```
	SEM_Init();

#if SDK_REMOTE_MODE
	// simulate button press
	SDK_TIM_AddInterrupt(&SDK_BTN_SetDown, SEM_BTN_PERIOD, true);

	// simulate command input
	SDK_TIM_AddDataInterrupt(&SDK_MAIN_ProcessCommand, "set interrupts 0", 100,
false);
	SDK_TIM_AddDataInterrupt(&SDK_MAIN_ProcessCommand, "set timeout 100", 100,  false);
	SDK_TIM_AddDataInterrupt(&SDK_MAIN_ProcessCommand, "set mode 2", 700,  false);
	SDK_TIM_AddDataInterrupt(&SDK_MAIN_ProcessCommand, "?", 1000, false);
#endif
}

void SDK_MAIN_ProcessCommand(const char* command)
{
	CMD_ProcessCommand(command);
}
```

Commands

Модуль для обработки поступивших команд, перенаправленных через main_wrapper из uart. В первую очередь вызывается функция **CMD_ProcessCommand**, вызывающая **CMD_ParseComand**. При успешном распознавании возвращаются вид и аргументы команды и только после происходит выполнение.

App/commands.h

```
/// TYPES ///
typedef enum eCmdType
{
	ECT_GetInfo,		// '?'
	ECT_SetMode,		// 'set mode X'
	ECT_SetTimeout,		// 'set timeout X'
	ECT_SetInterrupts,	// 'set interrupts X'
	ECT_Undefined
} eCmdType;

typedef struct CmdData
{
	eCmdType m_type;
	uint32_t m_arg;
} CmdData;

/// API ///
void CMD_ProcessCommand(const char* command);
void CMD_GetInfo();

CmdData CMD_ParseComand(const char* command);

bool CMD_ParseGetInfo(const char* command, int strSize, CmdData* data);
bool CMD_ParseSet(const char* command, int strSize, CmdData* data);
bool CMD_ParseSetMode(const char* command, int strSize, CmdData* data);
bool CMD_ParseSetTimeout(const char* command, int strSize, CmdData* data);
bool CMD_ParseSetInterrupts(const char* command, int strSize, CmdData* data);
```

App/commands.c

```
/// STATIC ///
static char* MapColorStateToName(eColorState color)
{
	switch(color)
```

```
	{
	case ECS_Green:
		return "Green";
	case ECS_Yellow:
		return "Yellow";
	case ECS_Red:
		return "Red";
	case ECS_BlinkingGreen:
		return "Blinking Green";
	}
	return "";
}
static char* MapModeToName(eSemaphoreMode mode)
{
	switch(mode)
	{
	case ESM_ProcessPress:
		return "Process";
	case ESM_IgnorePress:
		return "Ignore";
	}
	return "";
}
static char* MapBool(bool value)
{
	if(value)
	{
		return "true";
	}
	return "false";
}

/// API ///
void CMD_ProcessCommand(const char* command)
{
	CmdData data = CMD_ParseComand(command);
	switch(data.m_type)
	{
	case ECT_GetInfo:
		CMD_GetInfo();
		break;
	case ECT_SetMode:
		SEM_SetMode(data.m_arg);
		break;
	case ECT_SetTimeout:
		SEM_SetRedTimeout(data.m_arg);
		break;
	case ECT_SetInterrupts:
		SDK_UART_EnableInterrupts(data.m_arg);
		break;
	default:
		break;
	}
}
void CMD_GetInfo()
{
	SemaphoreState info = SEM_GetState();
	char buffer[128];

	sprintf(buffer, "\nColor: %s \nModeOnPress: %s \nRed timeout: %d \nInterrupts: %s
\n",
		MapColorStateToName(info.m_color), MapModeToName(info.m_mode),
```

```
                    info.m_redTimeout, MapBool(SDK_UART_IsInterruptible()));

        SDK_UART_Transmit((uint8_t*)buffer, strlen(buffer));
}

// parsing
CmdData CMD_ParseComand(const char* command)
{
        CmdData data;
        data.m_type = ECT_Undefined;
        data.m_arg = 0;

        size_t strSize = strlen(command);
        uint32_t commandStart = 0;

        // find first non-space symbol
        while (commandStart < strSize && isspace((int)command[commandStart]))
        {
                commandStart++;
        }
        if (commandStart == strSize)
        {
                return data;
        }
        command = command + commandStart;
        strSize -= commandStart;

        // try parse as different commands
        if (CMD_ParseGetInfo(command, strSize, &data))
        {
                return data;
        }
        if (CMD_ParseSet(command, strSize, &data))
        {
                return data;
        }

        return data;
}

bool CMD_ParseGetInfo(const char* command, int strSize, CmdData* data)
{
        char* pCh = strstr(command, "?");
        if (command && pCh == command &&
                (strSize == 1 || isspace((int)command[1])))
        {
                data->m_type = ECT_GetInfo;
                data->m_arg = 0;
                return true;
        }
        return false;
}

bool CMD_ParseSet(const char* command, int strSize, CmdData* data)
{
        char* pCh = strstr(command, "set");
        if (command && pCh == command &&
                strSize > 3 && isspace((int)command[3]))
        {
                uint32_t commandStart = 3;
                while (commandStart < strSize && isspace((int)command[commandStart]))
                {
```

```
                commandStart++;
            }
            if (commandStart == strSize)
            {
                return false;
            }
            command = command + commandStart;
            strSize -= commandStart;


            if (CMD_ParseSetMode(command, strSize, data))
            {
                return true;
            }
            if (CMD_ParseSetTimeout(command, strSize, data))
            {
                return true;
            }
            if (CMD_ParseSetInterrupts(command, strSize, data))
            {
                return true;
            }
        }
        return false;
}

bool CMD_ParseSetMode(const char* command, int strSize, CmdData* data)
{
        int value;
        uint32_t success = sscanf(command, "mode %d", &value);
        if (success == 1)
        {
            data->m_type = ECT_SetMode;
            data->m_arg = value;
            return true;
        }
        return false;
}
bool CMD_ParseSetTimeout(const char* command, int strSize, CmdData* data)
{
        int value;
        uint32_t success = sscanf(command, "timeout %d", &value);
        if (success == 1)
        {
            data->m_type = ECT_SetTimeout;
            data->m_arg = value;
            return true;
        }
        return false;
}
bool CMD_ParseSetInterrupts(const char* command, int strSize, CmdData* data)
{
        int value;
        uint32_t success = sscanf(command, "interrupts %d", &value);
        if (success == 1)
        {
            data->m_type = ECT_SetInterrupts;
            data->m_arg = value;
            return true;
        }
        return false;
}
```

Driver

Main

В функцию инициализации было добавлено создание двух UART

main.c

```
  HAL_Init();
  SystemClock_Config();

  MX_GPIO_Init();
  MX_IWDG_Init();
  MX_USART2_UART_Init();
  MX_USART3_UART_Init();

  SDK_MAIN_Wrapper();
```

UART

Для инициализации uart создается структура с характеристиками, описанными в инструментарии, и вызывается **HAL_UART_Init**

uart.c

```
void MX_USART2_UART_Init(void)
{

  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }

}
/* USART3 init function */
void MX_USART3_UART_Init(void)
{

  huart3.Instance = USART3;
  huart3.Init.BaudRate = 115200;
  huart3.Init.WordLength = UART_WORDLENGTH_8B;
  huart3.Init.StopBits = UART_STOPBITS_1;
  huart3.Init.Parity = UART_PARITY_NONE;
  huart3.Init.Mode = UART_MODE_TX_RX;
  huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart3.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart3) != HAL_OK)
  {
    Error_Handler();
  }

}
```

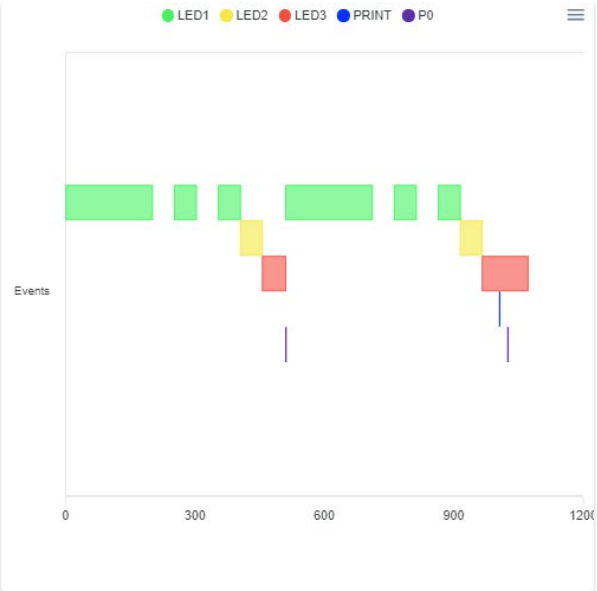
Выводы

Главной проблемой, приемлемого решения которой найти не удалось, оказалась невозможность отправки на UART каких-либо данных удаленно. В качестве примера на GitHub приведен как раз схожий случай, но на практике в данный момент нет соединенных друг с другом UART для возможности локальной посылки. В связи с этим поведение можно лишь примерно имитировать через отложенный ручной вызов команд.

Иных особых трудностей выполнение не вызывало. В процессе была освоены разные виды передачи через UART и обработка соответствующих прерываний.

Полный исходный код ЛР можно найти на github https://github.com/Old-Fritz/EmbededSystems

Результат работы:




```
/****** SDK-1.1M Trace start (06.04.2021 15:53:02) ******/
  1006.006: Uart:
Color: Red
ModeOnPress: Ignore
Red timeout: 100
Interrupts: false

  /***************** SDK-1.1M Trace stop ******************/
```