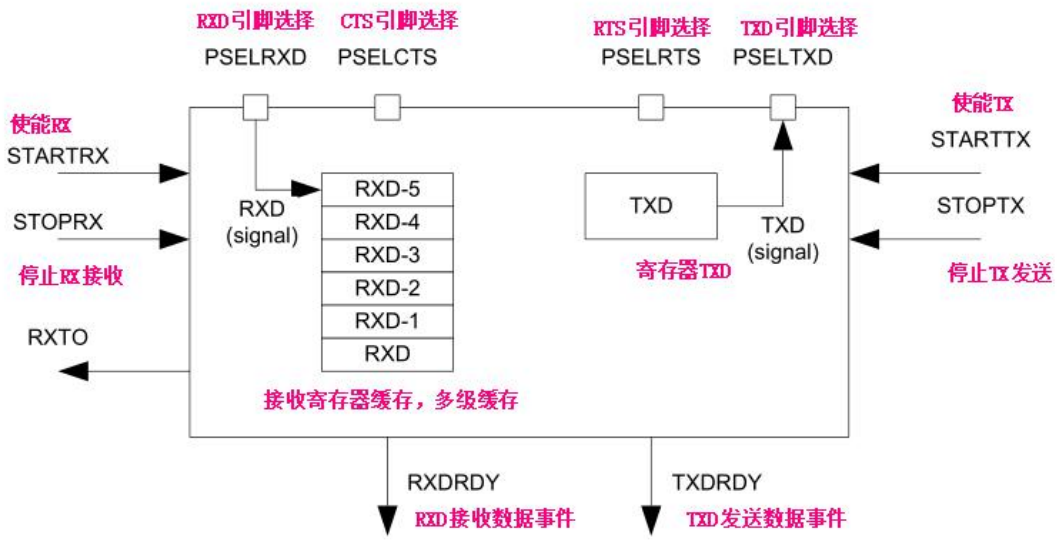
第九章 串口 UART 和 UARTE 外设应用

9.1 UART 和 UARTE 原理

9.1.1 UART 功能描述

串口 UART 也称为通用异步收发器。是各种处理器中常用了通信接口，在 nRF52 芯片中，UART 具有以下特点：

- 全双工操作
- 自动流控
- 奇偶校验产生第 9 位数据



◆硬件配置：

根据 PSELRXD、PSELCTS、PSELRTS 和 PSELTXD 寄存器的配置可以相应的将 RXD、CTS(发送清除、低有效)、TXD、RTS（发送请求、低有效）映射到物理的引脚上。如果这些寄存器的任意一个设为 0xffffffff，相关的 UART 信号就不会连接到任务物理引脚上。这四个寄存器及其配置只能在 UART 使能时可用，可以在芯片为系统 ON 模式时保持。为了在系统处于 OFF 关闭模式时通过 UART 确保引脚上的信号电平正确，必须按照 GPIO 外设中的说明在 GPIO 外设中配置引脚。如下表所示：

管脚	系统 ON 模式下使能 UART	系统 OFF 关闭模式下 GPIO 配置
PSELRXD	RXD 串口接收端	输入，无输出值
PSELCTS	流量控制送清除、低有效	输入，无输出值
PSELRTS	流量控制发送请求、低有效	输出，高电平输出 1
PSELTXD	TXD 串口发送端	输出，高电平输出 1

◆UART 发送：

通过触发 STARTTX 任务启动 UART 传输序列。

通过写入 TXD 寄存器来发送字节。 成功发送一个字节后，UART 将产生一个 TXDRDY 事件，之后可以将一个新字节写入 TXD 寄存器。

通过触发 STOPTX 任务立即停止 UART 传输序列。

如果启用了流量控制，则在 CTS 取消激活时将自动暂停传输，并在再次激活 CTS 时恢复。在 CTS 被停用时正在传输的字节将在传输暂停之前被完全传输。

◆UART 接收：

通过触发 STARTRX 任务启动 UART 接收序列。

UART 接收器连接了一个 FIFO，能够在数据被覆盖之前存储六个传入的 RXD 字节。 通过读取 RXD 寄存器从该 FIFO 中提取字节。 当从 FIFO 中提取一个字节时，FIFO 中待处理的新字节将被移动到 RXD 寄存器。 每次将新字节移入 RXD 寄存器时，UART 都会产生 RXDRDY 事件。

当启用流量控制时，当接收器 FIFO 中只有 4 个字节的空间时，UART 将禁用 RTS 信号。 因此，在重写数据之前，对应的发送器能够在 RTS 信号被去激活之后发送多达四个字节。 因此，为防止覆盖 FIFO 中的数据，对应的 UART 发送器必须确保在 RTS 线停用后在四个字节内停止发送数据。

当 FIFO 清空时，首先再次激活 RTS 信号，也就是说 CPU 读取 FIFO 中的所有字节时，当接收器通过 STOPRX 任务停止时，RTS 信号也将被禁用。在 RTS 信号被停用后，它们会立即连续发送。当该周期结束时，UART 将生成接收器超时事件（RXTO）。

为防止输入数据丢失，必须在每次 RXDRDY 事件后读取 RXD 寄存器一次。为了确保 CPU 可以通过 RXDRDY 事件寄存器检测所有输入的 RXDRDY 事件，必须在读取 RXD 寄存器之前清零 RXDRDY 事件寄存器。这样做的原因是允许 UART 将新字节写入 RXD 寄存器，因此在 CPU 读取（清空）RXD 寄存器后立即生成新事件。

◆UART 挂起：

UART 串口可以通过触发 SUPSPEND 寄存器任务来挂起。SUPSPEND 将会影响 UART 发送器和 UART 接收器，设置后使发生器停止发送，接收器停止接收。在 UART 挂起后，通过相应的触发 STARTTX 和 STARTRX 就可以重新开启发送和接收。

当触发 SUPSPEND 任务时，UART 接收器和触发 STOPRX 任务一样的工作。

在触发 SUPSPEND 任务后，正在进行的 TXD 字节传输将在 UART 挂起前完成。

◆错误条件：

在一个错误帧出现的情况下，如果再此帧中没有检测到有效的停止位会产生一个 ERROR 事件。另外，在中断时，如果 RXD 保持低电平超过一个数据帧长度时，也会产生一个 ERROR 事件。

◆流量控制：

nRF52 芯片的 UART 可以分为带流量控制和不带流量控制两种方式。不带流量控制时，不需要连接 CTS 和 RTS 两个管脚，可以视为两个管脚一值有效。

带流量控制时，RTS 引脚作为输出，由 UART 硬件模块自动控制。与接收寄存器的多级硬件缓冲 Buff 协调工作。比如在硬件缓冲已经接收满了 6 个字节的时，RTS 引脚就输出高电平的终止信号，当缓冲中的数据都被读出后回复有效信号（低电平）。

CTS 作为输入由外部输入。当 CTS 有效时（低电平）模块可以发送，当为无效时，模块自动暂停发送，并在 CTS 恢复有效时继续发送。

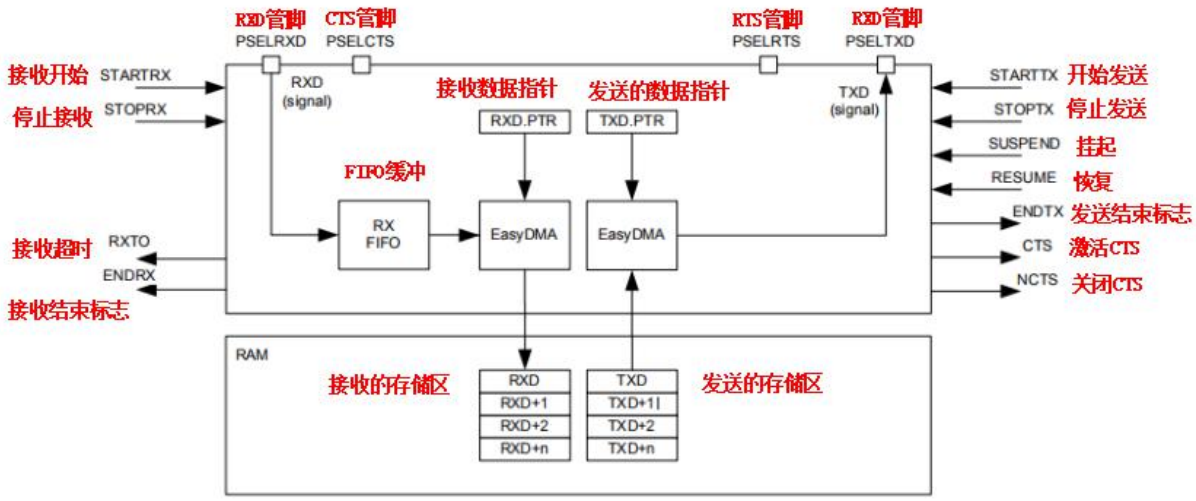
那么当 uart 模块的 rts 与 cts 交叉相接。 如果发送方发送太快，当接收方的接收硬件 buff 已经存满了字节后，接收方自动无效 RTS 信号，表示不能接收了。因为接收方 RTS 与发送方 CTS 相接，使得发送方的 CTS 也编程无效信号，于是发送方自动停止发送。这样就保证了接收方不会接收溢出。流量控制也就是体现在这里。

9.1.1 UARTE 功能介绍

UARTE 就是带有 EasyDMA 的通用异步接收器/发送器 UART。提供快速、全双工、异步的串行通信，内置流量控制（CTS，RTS）支持硬件，速率高达 1 Mbps。这里列出的是 UARTE 的主要功能：

- 全双工操作
- 自动硬件流控制
- 生成 9 位数据带奇偶校验
- EasyDMA
- 波特率高达 1 Mbps
- 在支持的事务之间返回 IDLE（使用 HW 流控制时）
- 一个停止位
- 最低有效位（LSB）优先

用于每个 UART 接口的 GPIO 可以从设备上的任何 GPIO 来选择并且独立地为可配置的。这使得能够在器件的引脚和有效地利用电路板空间和信号路由很大的灵活性。



寄存器	说明	寄存器	说明
PSEL.RXD	RXD 管脚选择	ENDTX	最后一个 TX 字节传输完成
PSEL.CTS	CTS 管脚选择	CTS	激活 CTS(设置低)。清除发送。
PSEL.RTS	RTS 管脚选择	NCTS	CTS 被停用(设置高)。不清除发送。
PSEL.TXD	TXD 管脚选择	RXTO	接收超时
STARTRX	开始串口接收	ENDRX	接收缓冲区被填满
STOPRX	停止串口接收	RX FIFO	接收 FIFO 缓冲
STARTTX	开始串口发送	RXD.PTR	接收的数据指针
STOPTX	停止串口发送	TXD.PTR	发送的数据指针
SUSPEND	挂起	TXD.MAXCNT	发送缓冲最大的存储字节
RESUME	恢复	RXD.MAXCNT	接收缓冲最大的存储字节

UARTE 实现 EasyDMA 读取和写入，并存入 RAM。如果 TXD.PTR 和 RXD.PTR 没有指向

数据 RAM 区，EasyDMA 传递可能导致 HardFault 或 RAM 损坏。.PTR 和.MAXCNT 寄存器是双缓冲的。他们可以在收到 RXSTARTED / TXSTARTED 事件后立即进行更新，并在接下来的 RX / TX 传送准备。所述 ENDRX / ENDTX 事件表示 EasyDMA 已完成访问分别在 RAM 中的 RX / TX 缓冲器。

◆UARTE 发送：

一个 UARTE 的发送的第一个步骤是存储的字节到发送缓冲器和配置 EasyDMA。这个过程是通过写起始地址到指针 TXD.PTR，并在 RAM 缓冲器放置 TXD.MAXCNT 大小的字节数来实现的。串口 UARTE 的发送是通过触发 STARTTX 任务开始，之后的每个字节在 TXD 线上发送时，会产生一个 TXDRDY 事件。当在 TXD 缓冲器中的所有字节（在 TXD.MAXCNT 寄存器中指定数目）已被传送时，UARTE 传输将自动结束，并且将产生一个 ENDTX 事件。

通过触发 STOPTX 任务来停止 UARTE 发送序列，当 UARTE 发射机已经停止，将产生一个 TXSTOPPED 事件。如果在 UARTE 发射机已经停下来但尚未产生 ENDTX 事件时，UARTE 将明确产生一个 ENDTX 事件，即使在 TXD 缓冲区中的所有字节（TXD.MAXCNT 寄存器中指定）还没有被发送。

如果启用了流量控制，则在 CTS 取消激活时将自动暂停传输，并在再次激活 CTS 时恢复。在 CTS 被停用时正在传输的字节将在传输暂停之前被完全传输。

◆UARTE 接收：

通过触发 STARTRX 任务启动 UARTE 接收器。UARTE 接收器使用 EasyDMA 将输入数据存储在 RAM 中的 RX 缓冲区中。RX 缓冲区位于 RXD.PTR 寄存器中指定的地址。RXD.PTR 寄存器是双缓冲的，可以在生成 RXSTARTED 事件后立即更新并为下一个 STARTRX 任务做好准备。 RX 缓冲区的大小在 RXD.MAXCNT 寄存器中指定，UARTE 在填充 RX 缓冲区时将生成 ENDRX 事件。

对于通过 RXD 线接收的每个字节，都将生成 RXDRDY 事件。在将相应的数据传输到数据 RAM 之前，可能会发生此事件。在 ENDRX 事件之后可以查询 RXD.AMOUNT 寄存器，以查看自上一次 ENDRX 事件以来有多少新字节已传输到 RAM 中的 RX 缓冲区。

通过触发 STOPRX 任务来停止 UARTE 接收器。UARTE 停止时会生成 RXTO 事件。 UARTE 将确保在生成 RXTO 事件之前生成即将发生的 ENDRX 事件。 这意味着 UARTE 将保证在 RXTO 之后不会生成 ENDRX 事件，除非重新启动 UARTE 或在生成 RXTO 事件后发出 FLUSHRX 命令。

重要提示：如果在 UARTE 接收器停止时尚未生成 ENDRX 事件，这意味着 RX FIFO 中的所有待处理内容都已移至 RX 缓冲区，则 UARTE 将显式生成 ENDRX 事件，即使 RX 缓冲区未满。 在这种情况下，将在生成 RXTO 事件之前生成 ENDRX 事件。

为了能够知道实际接收到 RX 缓冲区的字节数，CPU 可以在 ENDRX 事件或 RXTO 事件之后读取 RXD.AMOUNT 寄存器。只要在 RTS 信号被禁用后立即连续发送，UARTE 就可以在 STOPRX 任务被触发后接收最多四个字节。这是可能的，因为在 RTS 取消激活后，UARTE 能够在一段延长的时间内接收字节，该时间等于在配置的波特率上发送 4 个字节所需的时间。生成 RXTO 事件后，内部 RX FIFO 可能仍包含数据，要将此数据移至 RAM，必须触发 FLUSHRX 任务。

为确保此数据不会覆盖 RX 缓冲区中的数据，应在 FLUSHRX 任务被触发之前清空 RX 缓冲区或更新 RXD.PTR。为确保 RX FIFO 中的所有数据都移至 RX 缓冲区，RXD.MAXCNT 寄存器必须设置为 RXD.MAXCNT> 4，通过 STOPRX 强制停止的 UARTE 接收。即使 RX FIFO 为空或 RX 缓冲区未填满，UARTE 也会在完成 FLUSHRX 任务后生成 ENDRX 事件。 为了能够知道在这种情况下实际接收到 RX 缓冲区的字节数，CPU 可以在 ENDRX 事件之后读取 RXD.AMOUNT 寄存器。

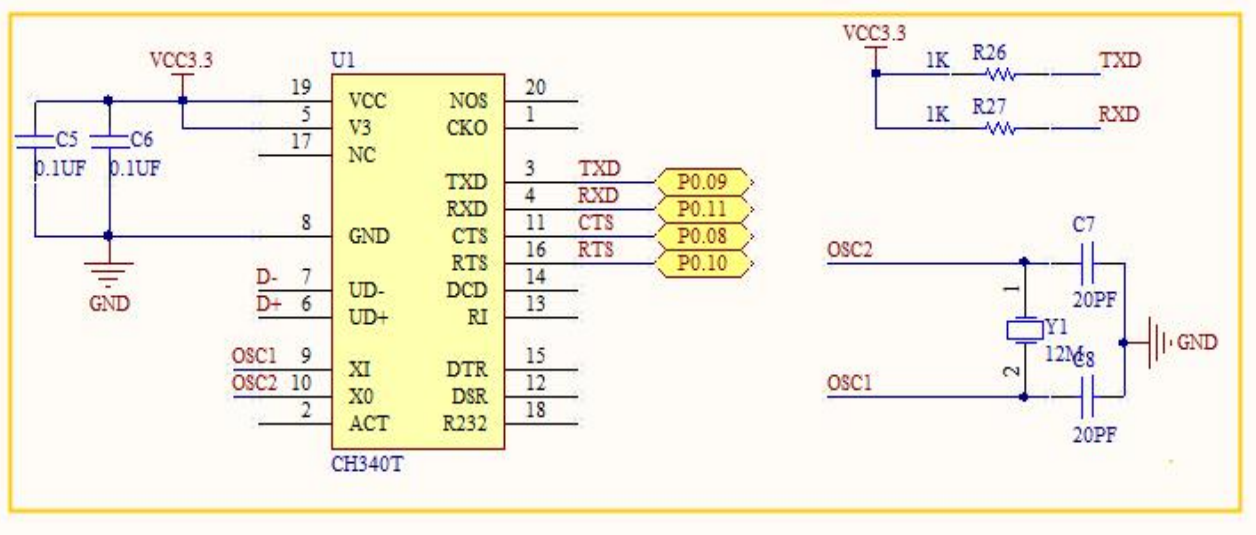
如果启用了 HW 流量控制，当接收器通过 STOPRX 任务停止或 UARTE 只能在其内部 RX FIFO 中接收 4 个字节时，RTS 信号将被禁用。禁用流量控制后，UARTE 将以与启用流量控制时相同的方式运行，但不会使用 RTS 线路。 这意味着当 UARTE 达到只能在其内部 RX FIFO 中接收四个字节的点时，不会产生任何信号。 内部 RX FIFO 填满时接收的数据将丢失。UARTE 接收器将处于最

低活动水平，并在停止时消耗最少的能量，即在通过 STARTRX 启动之前或通过 STOPRX 停止并且已生成 RXTO 事件之后。

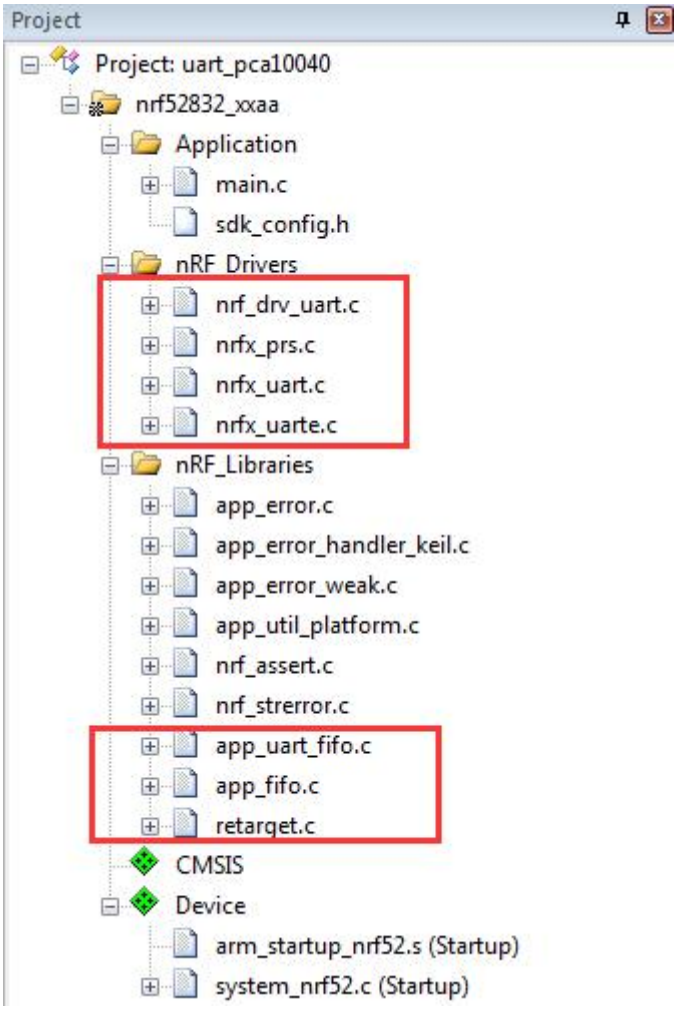
9.2 应用实例编写

9.2.1 串口 printf 输出

硬件连接方面，通过高质量芯片 CH340T 把串口信号转换成 usb 输出， TXD--接 P0.09 端 RXD--接 P0.11 端 CTS--接 P0.08 端 RTS--接 P0.10 端其中数据口 TXD 和 RXD 分别接 1K 电阻上拉，提高驱动能力，如下图所示。



在代码文件中，实验建立了一个演示历程，我们还是采用分层思想，直接通过官方提供的组件库进行编程。打开文件夹中的 uart 工程，添加 UART 的组件库如下图所示：



新增文件名称	功能描述
nrf_drv_uart.c	老版本串口驱动库
nrfx_prs.c	外设资源共享库
nrfx_uart.c	新版本 uart 串口兼容库
nrfx_uarte.c	新版本 uarte 串口兼容库
app_uart_fifo.c	uart 应用 fifo 的驱动库
app_fifo.c	应用 fifo 的驱动库
retarget.c	串口重定义文件

如上图所示：nrf_drv_uart.c 文件和 app_uart_fifi.c 文件是官方编写好的驱动库文件。这两个文件在后面编写带协议栈的 BLE 应用时，是可以直接用于其中配置串口功能的。因此理解这两个文件提供的 API 函数是利用组件编写串口的必由之路。

在 app_uart.h 文件中提供了两个关键的 uart 的初始化函数 APP_UART_FIFO_INIT 和 APP_UART_INIT，一个是带 FIFO 缓冲的初始化串口函数，一个是不带 FIFO 缓冲的初始化函数，一般情况下使用带软件缓冲的 FIFI 的函数，减小数据溢出错误的发生几率。配置代码具体如下：

```
01.  APP_UART_FIFO_INIT(&comm_params,        //串口参数
02.                     UART_RX_BUF_SIZE,//RX 缓冲大小
```

```
                                    UART_TX_BUF_SIZE,//TX 缓冲大小
                                    uart_error_handle,//错误处理
                                    APP_IRQ_PRIORITY_LOW,//中断优先级
                                    err_code);//配置串口
```

这个函数配置参数较多，首先是&comm_params 参数，这个参数配置了串口管脚，流量控制，波特率等关键参数。这些参数通过一个结构体进行定义，参数作为结构体的内容，代码如下所示：

```
    const app_uart_comm_params_t comm_params =
      {
          RX_PIN_NUMBER,
          TX_PIN_NUMBER,
          RTS_PIN_NUMBER,
          CTS_PIN_NUMBER,//串口管脚配置
          APP_UART_FLOW_CONTROL_DISABLED,//流控设置
          false,
          UART_BAUDRATE_BAUDRATE_Baud115200//波特率
      };//设置串口参数

```

RX_PIN_NUMBER、TX_PIN_NUMBER、RTS_PIN_NUMBER、CTS_PIN_NUMBER 四个配置串口的硬件管脚，带流控下四个管脚都需要宏定义 IO 端口。如果是不带流控的模式下，只需要配置 RX_PIN_NUMBER 和 RTX_PIN_NUMBER 两个管脚。

设置 APP_UART_FLOW_CONTROL_DISABLED 关掉流控。

设置中断优先级为低。

设置 UART_BAUDRATE_BAUDRATE_Baud115200 波特率为 115200。

那么这些参数如何赋值给串口的寄存器？这就需要深入 APP_UART_FIFO_INIT 内部研究了，进入函数内部：

```
#define APP_UART_FIFO_INIT(P_COMM_PARAMS, RX_BUF_SIZE, TX_BUF_SIZE,
    EVT_HANDLER, IRQ_PRIO, ERR_CODE) \
    do
    {
        app_uart_buffers_t buffers;
        static uint8_t        rx_buf[RX_BUF_SIZE];
        static uint8_t        tx_buf[TX_BUF_SIZE];
        buffers.rx_buf          = rx_buf;
        buffers.rx_buf_size = sizeof (rx_buf);
        buffers.tx_buf          = tx_buf;
        buffers.tx_buf_size = sizeof (tx_buf);
        ERR_CODE = app_uart_init(P_COMM_PARAMS, &buffers, EVT_HANDLER,
    IRQ_PRIO);
    }
```

函数内部申请两个软件 BUF 缓冲空间提供给 RX 和 TX。然后调用函数 app_uart_init 进行串口的初始化。继续深入 app_uart_init 函数内部，这段函数是官方给的驱动组件库，读者不需要单独配置或者修改，只需要直接调用：

```
uint32_t app_uart_init(const app_uart_comm_params_t * p_comm_params,
                                    app_uart_buffers_t *        p_buffers,
```

```
                                        app_uart_event_handler_t event_handler,
                                        app_irq_priority_t       irq_priority)
{
    uint32_t err_code;

    m_event_handler = event_handler;

    if (p_buffers == NULL)
    {
        return NRF_ERROR_INVALID_PARAM;
    }

    // 配置 RX buffer 缓冲.
    err_code = app_fifo_init(&m_rx_fifo, p_buffers->rx_buf, p_buffers->rx_buf_size);
    if (err_code != NRF_SUCCESS)
    {
        // Propagate error code.
        return err_code;
    }

    // 配置 TX buffer 缓冲
    err_code = app_fifo_init(&m_tx_fifo, p_buffers->tx_buf, p_buffers->tx_buf_size);
    if (err_code != NRF_SUCCESS)
    {
        // Propagate error code.
        return err_code;
    }
   //配置串口的管脚，波特率，流控等特性
    nrf_drv_uart_config_t config = NRF_DRV_UART_DEFAULT_CONFIG;
    config.baudrate = (nrf_uart_baudrate_t)p_comm_params->baud_rate;
    config.hwfc = (p_comm_params->flow_control
      ==APP_UART_FLOW_CONTROL_DISABLED) ?
            NRF_UART_HWFC_DISABLED : NRF_UART_HWFC_ENABLED;
    config.interrupt_priority = irq_priority;
    config.parity = p_comm_params->use_parity ? NRF_UART_PARITY_INCLUDED :
  NRF_UART_PARITY_EXCLUDED;
    config.pselcts = p_comm_params->cts_pin_no;
    config.pselrts = p_comm_params->rts_pin_no;
    config.pselrxd = p_comm_params->rx_pin_no;
    config.pseltxd = p_comm_params->tx_pin_no;

    err_code = nrf_drv_uart_init(&config, uart_event_handler);

    if (err_code != NRF_SUCCESS)
```

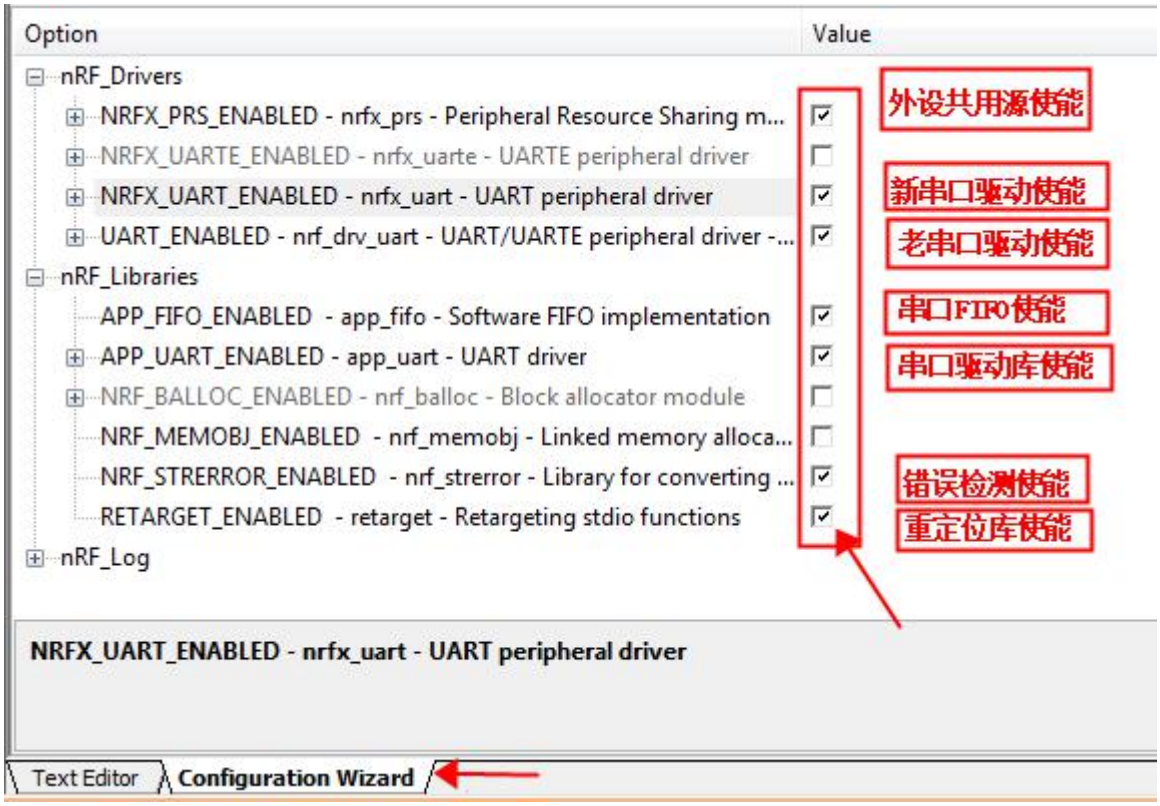
```
46.     {
47.         return err_code;
48.     }
49.     nrf_drv_uart_rx_enable();
50.     return nrf_drv_uart_rx(rx_buffer,1);
51. }
```

这个函数 app_uart_init 该函数介绍如下所示：

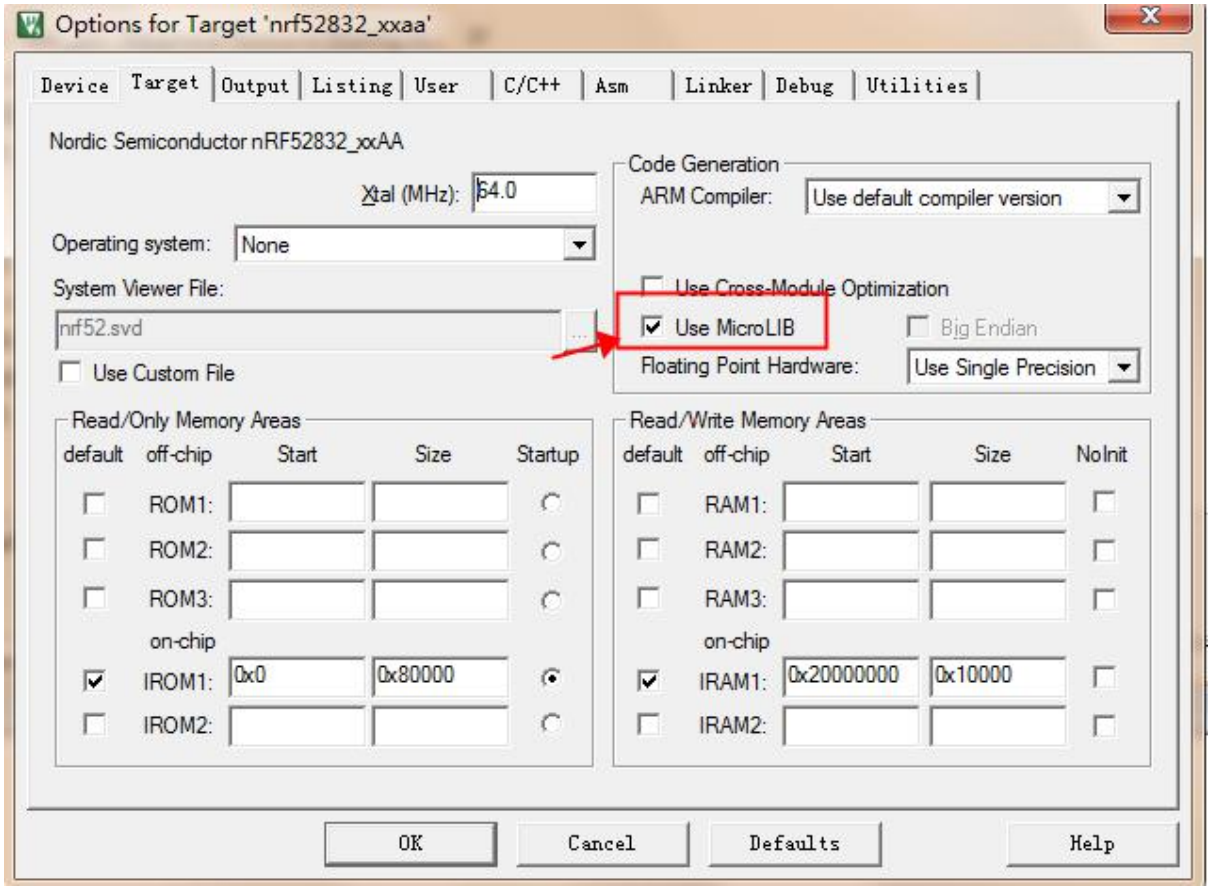
函数：uint32_t app_uart_init(const app_uart_comm_params_t * p_comm_params, app_uart_buffers_t * p_buffers, app_uart_event_handler_t error_handler, app_irq_priority_t irq_priority);
*功能：用于初始化 UART 模块的函数。当需要 UART 模块实例时，使用此初始化。注意：通常应该使用 APP_UART_INIT()或 APP_UART_INIT_FIFO()宏执行单个初始化，这取决于 UART 是否应该使用 FIFO，因为这将分配 UART 模块所需的缓冲区(包括正确对齐缓冲区)。
* 参数 p_comm_params Pin 和通信参数。 p_buffers RX 和 TX 缓冲区，NULL 是 FIFO 不使用。 error_handler 错误处理函数，以便在发生错误时调用。 irq_priority 中断优先级。
*返回值： NRFX_SUCCESS 如果初始化成功 * 返回值： NRF_ERROR_INVALID_LENGTH 如果所提供的缓冲区不是 2 的幂。 * 返回值： NRF_ERROR_NULL 如果提供的缓冲区之一是空指针。 当启用硬件流控制时，UART 模块在注册时将以下错误传播给调用者。当不使用硬件流控制时，就不会发生这些错误。 * 返回值：NRF_ERROR_INVALID_STATE 在将 UART 模块注册为用户时，GPIOTE 模块没有处于有效状态。 * 返回值：NRF_ERROR_INVALID_PARAM 当将 UART 模块注册为用户时，UART 模块提供了一个无效的回调函数。或者 p_uart_uid 指向的值不是有效的 GPIOTE 号。 * 返回值：NRF_ERROR_NO_MEM GPIOTE 模块已达到最大用户数。

工程编译之前，还需要在 sdk_config.h 文件中配置使能下面几个功能：

外设共用源的使能、新串口驱动使能、老板串口驱动使能、串口 FIFO 缓冲库使能、串口驱动库使能、错误检测使能、重定位库使能。可以直接打开 sdk_config.h 文件中的 configuration Wizard 向导进行勾选，如下图所示。如果 sdk_config.h 文件中没有相关配置，可以拷贝历程中的库使能的对应源码到配置文件中。



本例使用 printf()函数是格式化输出函数，一般用于向标准输出设备按规定格式输出信息。本格式的输出，是 c 语言中产生格式化输出的函数（在 stdio.h 中定义）。用于向终端（显示器、控制台等）输出字符。格式控制由要输出的文字和数据格式说明组成。要输出的文字除了可以使用字母、数字、空格和一些数字符号以外，还可以使用一些转义字符表示特殊的含义。因此，使用之前除了要使能重定向以外，还需要在 options for Target 选项卡中，选择 Target 后，勾选 Use MicroLIB 库，如下图所示：



那么主函数就是十分的简单了，直接调用我们串口的驱动库函数进行配置，输出采用 printf 函数打印输出，代码如下所示：

```
/******************** (C) COPYRIGHT 2019 青风电子 ********************
 * 文件名  ：main
 * 实验平台：青云 nRF52832 开发板
 * 描述    ：串口 printf 输出
 * 作者    ：青风
 * 论坛    ：www.qfv8.com
 ********************************************************************************/
int main(void)
{
    LEDS_CONFIGURE(LEDS_MASK);
    LEDS_OFF(LEDS_MASK);
    uint32_t err_code;
    const app_uart_comm_params_t comm_params =
      {
          RX_PIN_NUMBER,
          TX_PIN_NUMBER,
          RTS_PIN_NUMBER,
          CTS_PIN_NUMBER,//串口管脚配置
          APP_UART_FLOW_CONTROL_DISABLED,//流控设置
          false,
```

```
            UART_BAUDRATE_BAUDRATE_Baud115200//波特率
        };//设置串口参数

    APP_UART_FIFO_INIT(&comm_params,        //串口参数
                       UART_RX_BUF_SIZE,//RX 缓冲大小
                       UART_TX_BUF_SIZE,//TX 缓冲大小
                       uart_error_handle,//错误处理
                       APP_IRQ_PRIORITY_LOW,//中断优先级
                       err_code);//配置串口

    APP_ERROR_CHECK(err_code);

  while (1)
    {
      LEDS_INVERT(LEDS_MASK);
     printf(" 2017.10.1  青风!\r\n");//答应输出
       nrf_delay_ms(500);
   }
}
```

实验下载到青云 nRF52832 开发板后连接 usb 转串口端，如下图所示，然后打开串口调试助手，实验现象如下，指示灯闪亮表示串口输出进行中，打开串口调试助手，输出要求的内容：

9.2.2 串口输入与回环

上一节通过 printf 打印输出数据，这一节讲使用官方组件库来输入和输出数据。对应串口输

出和输入，官方的组件库提供了两个函数，分别为:

函数 app_uart_get 该函数介绍如下所示：

函数：uint32_t app_uart_get(uint8_t * p_byte);
*功能：为从 UART 串口获取数据。这个函数将从 RX 缓冲区获取下一个字节。如果 RX 缓冲区为空，将返回一个错误代码，app_uart 模块将在接收添加到 RX 缓冲区的第一个字节时生成一个事件。
* 参数 p_byte 指针指向下一个接收字节存放的地址。
* 返回值： NRF_SUCCESS 如果收到成功收到字节，则返回成功。 * 返回值：NRF_ERROR_NOT_FOUND 如果再 RX buffer 中没有发现收到数据，则返回 NRF_ERROR_NOT_FOUND。

函数 app_uart_put 该函数介绍如下所示：

函数：uint32_t app_uart_put(uint8_t byte);
*功能：通过串口输出字节。
* 参数[in] byte 串口传输的字节。
* 返回值： NRF_SUCCESS 如果通过 TX 缓冲把字节发送出去，则返回成功。 * 返回值：NRF_ERROR_NO_MEM 如果在 TX 缓冲中没有更多的空间。常用在流控控制中。 * 返回值：NRF_ERROR_INTERNAL 如果串口驱动报错。

使用上面两个函数，下面写一个串口回环测试，通过串口助手发送数据，通过串口接收后，发回串口助手。下面串口回环测试代码如下：

```
40. static void uart_loopback_test()
41. {
42. 
43.     while (true)
44.         {
45.             uint8_t cr;
46.             while(app_uart_get(&cr) != NRF_SUCCESS);//接收数据
47.             while(app_uart_put(cr) != NRF_SUCCESS);//把接收的数据发出
48. 
49.             if (cr == 'q' || cr == 'Q')
50.             {
51.                 printf(" \n\rExit!\n\r");//结束
52.                 while (true)
53.                 {
54.                     // Do nothing.
55.                 }
56.             }
57.         }
```

```
58.
59. }
```

主函数的编写主要就是配置串口参数，设串口管脚，比特率，流控等参数设置：

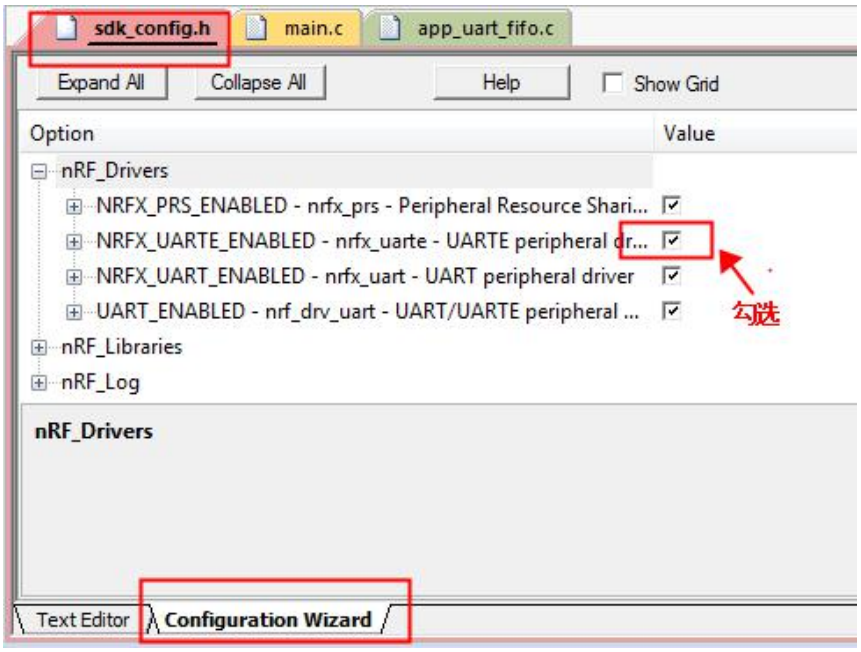
```
01. int main(void)
02. {
03.     LEDS_CONFIGURE(LEDS_MASK);
04.     LEDS_OFF(LEDS_MASK);
05.     uint32_t err_code;
06.     const app_uart_comm_params_t comm_params =
07.       {
08.           RX_PIN_NUMBER,//RX 管脚
09.           TX_PIN_NUMBER,//TX 管脚
10.           RTS_PIN_NUMBER,//RTS 管脚
11.           CTS_PIN_NUMBER,//CTS 管脚
12.           APP_UART_FLOW_CONTROL_DISABLED,//关掉流控
13.           false,
14.           UART_BAUDRATE_BAUDRATE_Baud115200//设置比特率
15.       };
16.
17.     APP_UART_FIFO_INIT(&comm_params,
18.                        UART_RX_BUF_SIZE,//设置 RX 缓冲
19.                        UART_TX_BUF_SIZE,//设置 TX 缓冲
20.                        uart_error_handle,
21.                        APP_IRQ_PRIORITY_LOW,
22.                        err_code);
23.
24.     APP_ERROR_CHECK(err_code);
25.     while (1)
26.     {
27.         uart_loopback_test();//调用回环
28.     }
29. }
```

编译程序后下载到青风 nrf52832 开发板内。打开串口调试助手，选择开发板串口端号，设置波特率为 115200，数据位为 8，停止位为 1。在发送区写入任何字符或者字符串，比如发送 nrf52832，点击发送后，接收端会接收到相同的字符或者字符串，如图接收到了 nrf52832。当输入字符 'q' 或者 'Q'，会输出 Exit!结束测试。

9.2.3 UARTE 串口中断

UARTE 和 UARTE 共享大部分寄存器设备，区别就是 UARTE 通过 EasyDMA 参与到数据的收发上。DMA 是 Direct Memory Access 的英文简写，直接内存存取。可以不需要 CPU 参与。普通传输中，CPU 需要从源位置把每一片段的数据复制到目的寄存器，然后把它们再次写回到新的地方。在这个时间中，CPU 对于其他的工作来说就无法使用。

EasyDMA 使得外围设备可以通过 DMA 控制器直接访问内存，与此同时，CPU 可以继续执行程序。EasyDMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 EasyDMA 来实行和完成。所有我们需要在 sdk_config.h 文件室内 UARTE 功能，如下图所示：



我们在初始化串口的时候设置带 FIFO 缓冲的串口，在 APP_UART_FIFO_INIT 函数中，申请一个 uart_Interrupt_handle 中断回调处理函数，具体代码如下所示：

```
01. int main(void)
02. {
03.     uint32_t err_code;
04.     const app_uart_comm_params_t comm_params =
05.       {
06.           RX_PIN_NUMBER,
07.           TX_PIN_NUMBER,
08.           RTS_PIN_NUMBER,
09.           CTS_PIN_NUMBER,
10.           APP_UART_FLOW_CONTROL_DISABLED,
11.           false,
12.           UART_BAUDRATE_BAUDRATE_Baud115200
13.       };
14.
15.     APP_UART_FIFO_INIT(&comm_params,
16.                        UART_RX_BUF_SIZE,
17.                        UART_TX_BUF_SIZE,
18.                        uart_Interrupt_handle,//声明一个中断处理函数
19.                        APP_IRQ_PRIORITY_LOW,
20.                        err_code);
21.
22.     APP_ERROR_CHECK(err_code);
23.     while (1)
24.     {
25.     }
26. }
```

注意：uart_Interrupt_handle 仅仅是中断回调处理函数，并不是中断子函数。我们需要进入代码深入理解中断函数和中断回调函数之间的关系。后面的外设代码中，库函数编程中常用中断回调处理函数来进行编程，基本原理也是类似的。

首先，开启中断前，需要对中断进行使能。进入 APP_UART_FIFO_INIT 函数中，FIFO 和不带 FIFO 的串口初始化唯一的区别就是带了 FIFO 缓冲，这个在 printf 一节已经讲述，在函数中调用了 app_uart_init 函数：

```
147 #define APP_UART_FIFO_INIT(P_COMM_PARAMS, RX_BUF_SIZE, TX_BUF_SIZE, EVT_HANDLER, IRQ_PRIO, ERR_CODE) \
148     do                                                                                               \
149     {                                                                                                \
150         app_uart_buffers_t buffers;                                                                  \
151         static uint8_t     rx_buf[RX_BUF_SIZE];                                                      \
152         static uint8_t     tx_buf[TX_BUF_SIZE];                                                      \
153                                                                                                      \
154         buffers.rx_buf      = rx_buf;                                                                \
155         buffers.rx_buf_size = sizeof (rx_buf);                                                       \
156         buffers.tx_buf      = tx_buf;                                                                \
157         buffers.tx_buf_size = sizeof (tx_buf);                                                       \
158         ERR_CODE = app_uart_init(P_COMM_PARAMS, &buffers, EVT_HANDLER, IRQ_PRIO);                    \
159     } while (0)
```

继续把 app_uart_init 函数进行展开，函数内部调用了 nrf_drv_uart_init 函数，该函数会触发一个 uart_event_handler 事件派发函数，这个派发函数会在后面用于中断事件的分配：

```
161
162     err_code = nrf_drv_uart_init(&app_uart_inst, &config, uart_event_handler);
163     VERIFY_SUCCESS(err_code);
164     m_rx_ovf = false;
165
166     // Turn on receiver if RX pin is connected
167     if (p_comm_params->rx_pin_no != UART_PIN_DISCONNECTED)
168     {
169         return nrf_drv_uart_rx(&app_uart_inst, rx_buffer,1);
170     }
171     else
172     {
173         return NRF_SUCCESS;
174     }
175 }
```

nrf_drv_uart_init 函数内部继续展开，这里就通过配置函数勾选 UARTE 和 UART 的使能，来区分是使用 UARTE 还是 UART 功能。如果同时勾选了 UARTE 和 UART，则默认是使用 UARTE 功能，这是会调用 nrfx_uarte_init 初始化函数对 UARTE 进行初始化，代码如下所示：

```
104 #endif // defined(NRF_DRV_UART_WITH_UART)
105
106 ret_code_t nrf_drv_uart_init(nrf_drv_uart_t const *        p_instance,
107                              nrf_drv_uart_config_t const * p_config,
108                              nrf_uart_event_handler_t      event_handler)
109 {
110     uint32_t inst_idx = p_instance->inst_idx;
111     m_handlers[inst_idx] = event_handler;
112     m_contexts[inst_idx] = p_config->p_context;
113
114 #if defined(NRF_DRV_UART_WITH_UARTE) && defined(NRF_DRV_UART_WITH_UART)
115     nrf_drv_uart_use_easy_dma[inst_idx] = p_config->use_easy_dma;
116 #endif
117
118     nrf_drv_uart_config_t config = *p_config;
119     config.p_context = (void *)inst_idx;
120
121     ret_code_t result = 0;
122     if (NRF_DRV_UART_USE_UARTE)
123     {
124         result = nrfx_uarte_init(&p_instance->uarte,
125                                  (nrfx_uarte_config_t const *)&config,
126                                  event_handler ? uarte_evt_handler : NULL);
127     }
128     else if (NRF_DRV_UART_USE_UART)
129     {
130         result = nrfx_uart_init(&p_instance->uart,
131                                 (nrfx_uart_config_t const *)&config,
132                                 event_handler ? uart_evt_handler : NULL);
133     }
134     return result;
135 }
136
```

nrfx_uarte_init 函数内部，通过调用函数 interrupts_enable 和 nrf_uarte_enable 对串口中断进行了使能。也就是打开了串口中断。代码如下图所示：

```
225
226     if (p_cb->handler)
227     {
228         interrupts_enable(p_instance, p_config->interrupt_priority);
229     }
230
231     nrf_uarte_enable(p_instance->p_reg);
232     p_cb->rx_buffer_length           = 0;
233     p_cb->rx_secondary_buffer_length = 0;
234     p_cb->tx_buffer_length           = 0;
235     p_cb->state                      = NRFX_DRV_STATE_INITIALIZED;
236     NRFX_LOG_WARNING("Function: %s, error code: %s.",
237                      __func__,
238                      NRFX_LOG_ERROR_STRING_GET(err_code));
239     return err_code;
240 }
```

我们打开中断后，就等待中断处理的发生，一旦出现串口中断，进进入串口中断处理函数 uart_irq_handler 中，中断处理函数中，首先通过检测事件类型，来分配事件 event type:

main.c | app_uart.h | app_uart_fifo.c | nrf_drv_uart.c | nrfx_uarte.c | nrfx_uart.c | nrfx_uart.

```
501     NRFX_LOG_INFO("RX transaction aborted.");
502 }
503
504 static void uarte_irq_handler(NRF_UARTE_Type *        p_uarte,
505                               uarte_control_block_t * p_cb)
506 {
507     if (nrf_uarte_event_check(p_uarte, NRF_UARTE_EVENT_ERROR))      判断申请的事件
508     {
509         nrfx_uarte_event_t event;
510
511         nrf_uarte_event_clear(p_uarte, NRF_UARTE_EVENT_ERROR);
512
513         event.type                    = NRFX_UARTE_EVT_ERROR;        <-- 分配事件类型
514         event.data.error.error_mask   = nrf_uarte_errorsrc_get_and_clear(p_uarte);
515         event.data.error.rxtx.bytes   = nrf_uarte_rx_amount_get(p_uarte);
516         event.data.error.rxtx.p_data  = p_cb->p_rx_buffer;
517
518         // Abort transfer.
519         p_cb->rx_buffer_length = 0;
520         p_cb->rx_secondary_buffer_length = 0;
521
522         p_cb->handler(&event, p_cb->p_context);
523     }
524     else if (nrf_uarte_event_check(p_uarte, NRF_UARTE_EVENT_ENDRX))      判断申请的事件
525     {
526         nrf_uarte_event_clear(p_uarte, NRF_UARTE_EVENT_ENDRX);
527         size_t amount = nrf_uarte_rx_amount_get(p_uarte);
528         // If the transfer was stopped before completion, amount of transfered bytes
529         // will not be equal to the buffer length. Interrupted transfer is ignored.
530         if (amount == p_cb->rx_buffer_length)
531         {
532             if (p_cb->rx_secondary_buffer_length)
533             {
534                 uint8_t * p_data = p_cb->p_rx_buffer;
535                 nrf_uarte_shorts_disable(p_uarte, NRF_UARTE_SHORT_ENDRX_STARTRX);
536                 p_cb->rx_buffer_length = p_cb->rx_secondary_buffer_length;
537                 p_cb->p_rx_buffer = p_cb->p_rx_secondary_buffer;
538                 p_cb->rx_secondary_buffer_length = 0;
539                 rx_done_event(p_cb, amount, p_data);      <-- 分配事件类型
540             }
541             else
542             {
543                 p_cb->rx_buffer_length = 0;
544                 rx_done_event(p_cb, amount, p_cb->p_rx_buffer);
545             }
546         }
547     }
548
549     if (nrf_uarte_event_check(p_uarte, NRF_UARTE_EVENT_RXTO))
```

在串口初始化中断 uart_event_handler 事件派发函数，这个派发函数根据分配的事件类型来分配中断事件处理类型：

```
main.c | app_uart.h | app_uart_fifo.c | nrf_drv_uart.c | nrfx_uarte.c | nrfx_uart.c | nrfx_uart.h
63  static app_fifo_t                  m_rx_fifo;                          /**< RX FIFO buffer fo
64  static app_fifo_t                  m_tx_fifo;                          /**< TX FIFO buffer fo
65
66  static void uart_event_handler(nrf_drv_uart_event_t * p_event, void* p_context)
67  {
68      app_uart_evt_t app_uart_event;
69      uint32_t err_code;
70
71      switch (p_event->type)
72      {
73          case NRF_DRV_UART_EVT_RX_DONE:
74              // Write received byte to FIFO.
75              err_code = app_fifo_put(&m_rx_fifo, p_event->data.rxtx.p_data[0]);
76              if (err_code != NRF_SUCCESS)
77              {
78                  app_uart_event.evt_type           = APP_UART_FIFO_ERROR;
79                  app_uart_event.data.error_code    = err_code;
80                  m_event_handler(&app_uart_event);
81              }
82              // Notify that there are data available.
83              else if (FIFO_LENGTH(m_rx_fifo) != 0)
84              {
85                  app_uart_event.evt_type = APP_UART_DATA_READY;
86                  m_event_handler(&app_uart_event);
87              }
88
89              // Start new RX if size in buffer.
90              if (FIFO_LENGTH(m_rx_fifo) <= m_rx_fifo.buf_size_mask)
91              {
92                  (void)nrf_drv_uart_rx(&app_uart_inst, rx_buffer, 1);
93              }
94              else
95              {
96                  // Overflow in RX FIFO.
97                  m_rx_ovf = true;
98              }
99
100             break;
101
102         case NRF_DRV_UART_EVT_ERROR:
103             app_uart_event.evt_type                 = APP_UART_COMMUNICATION_ERROR;
104             app_uart_event.data.error_communication = p_event->data.error.error_mask;
105             (void)nrf_drv_uart_rx(&app_uart_inst, rx_buffer, 1);
106             m_event_handler(&app_uart_event);
107             break;
108
109         case NRF_DRV_UART_EVT_TX_DONE:
110             // Get next byte from FIFO.
111             if (app_fifo_get(&m_tx_fifo, tx_buffer) == NRF_SUCCESS)
```

所有在 uart_Interrupt_handle 中断回调处理函数中，触发对应的事件做响应的处理，本例将演示使用 UARTE 功能触发一个串口中断事件，实现串口的接收和发送中断功能。在组件库中，对应串口中断事件处理的类型使用了一个 app_uart_evt_type_t 结构体进行了定义。这个结构体定义如下代码所示：

```
01. typedef enum
02. {
03.     APP_UART_DATA_READY,            /*表示已接收到 UART 数据的事件。数据在 FIFO
04.                                     中可用，可以使用函数 app_uart_get 获取数据。*/
05.     APP_UART_FIFO_ERROR,            /*app_uart 模块使用的 FIFO 模块中出现错误。FIFO
06.                                     错误代码存储在 app_uart_evt_t.data 中的 error_code 字段。*/
07.     APP_UART_COMMUNICATION_ERROR, /*接收过程中发生通信错误。错误存储在
08.                                     app_uart_evt_t.data 中的 error_communication 字段。 */
09.     APP_UART_TX_EMPTY,              /*一个发送事件，指示 UART 已完成 TX FIFO 中所
10.                                     有可用数据的传输。*/
11.     APP_UART_DATA,                  /*一个接收事件指示 UART 数据已被接收，数据出现
12.                                     在 data 数据字段中。此事件只在未配置 FIFO 时使用。*/
13. } app_uart_evt_type_t;
```

其中，对应发送中断事件：中断事件处理类型为 APP_UART_TX_EMPTY 示 UART 已完成 TX

FIFO 中所有可用数据的传输。

对应接收中断事件：使用了 RX FIFO 缓冲 事件类型为 APP_UART_DATA_READY

不使用了 RX FIFO 缓冲事件类型为 APP_UART_DATA

因为我们初始化的是带 FIFO 缓冲的串口，引出触发的接收和发送中断事件分别为：

APP_UART_DATA_READY 事件和 APP_UART_TX_EMPTY 事件。

我们通过串口助手发送一个数据给处理器，当接芯片收完成后，会产生接收事件，该事件里，我们把接收缓冲里的数据放入&rx 数组。然后我们通过串口 printf 来打印该数组内的数据，print 打印是串口发送，产生发送事件，这时我们点亮一个 LED 灯表示发送成功，同时串口助手也会收到对应的数据。

```
01. void uart_Interrupt_handle(app_uart_evt_t * p_event)
02. {
03.   uint8_t RX;
04.     if (p_event->evt_type == APP_UART_COMMUNICATION_ERROR)
05.     {
06.         APP_ERROR_HANDLER(p_event->data.error_communication);
07.     }
08.     else if (p_event->evt_type == APP_UART_FIFO_ERROR)
09.     {
10.         APP_ERROR_HANDLER(p_event->data.error_code);
11.     }
12.
13.          //串口接收事件
14.     else if (p_event->evt_type == APP_UART_DATA_READY)
15.     {
16.      //从 FIFO 中读取数据
17.        app_uart_get(&RX);
18.      //串口发送输出数据
19.        printf("%c",RX);
20.     }
21.   //串口发送完成事件，当 printf 发送成功后
22.     else if (p_event->evt_type == APP_UART_TX_EMPTY)
23.     {
24.         nrf_gpio_pin_toggle(LED_1);
25.     }
26. }
```

编译程序后下载到青风 nrf52832 开发板内。打开串口调试助手，选择开发板串口端号，设置波特率为 115200，数据位为 8，停止位为 1。在发送区写入任何字符或者字符串，比如发送 qfv8.com，点击发送后，接收端会接收到相同的字符或者字符串，如图接收到了 qfv8.com。如果正确收到数据，LED1 灯也会点亮。