Application Note
ZHCACR3 – 5. 2023

MSPM0 UART 通信中 DMA 和 Ring Buffer 环形缓冲的应用

Leon Yan *China Central FAE Team*

ABSTRACT

全新一代的 MSPM0 微控制器基于 Cortex M0+内核，主频覆盖 24MHz 到 80MHz，提供了丰富的配置和多种增强功能，在内部的模拟接口上，最高达到 4MHz 采样率的 12bit ADC，以及 12bit 的 1MHz DAC，为数据采集和模拟输出提供了高性能的保障。多个内部集成的 OPA 运放节省了外部器件，降低了 PCB 空间占有率。内部的多路 DMA 通道，为多个外设、存储的数据传输提供了高速通路，降低 MCU 内核的占有率。新引入的 Event 事件系统带来了更加灵活的事件 Post 和 Subscribe 机制，对于各个模块、外设的相互关联带来了更多的可选择性和灵活性。

本应用文档中针对应用中最常见的 UART 通信，根据不同的应用场景会面临不同的数据格式、波特率、数据长度，尤其是高码率的情况下的数据收发，带来了挑战，如何有效的利用 MCU 内部模块和有限的 CPU 资源，在本文中描述了 DMA 和 Ring Buffer 环形缓冲结合的灵活方式，有效的解决了此类问题，在降低功耗和 CPU 负载上达到了平衡。

Contents

Figures

1. UART 串口在嵌入式系统中的应用

在大部分 MCU 应用场合下，都会需要通过接口进行数据传输交互，常见的接口有 UART、I2C、SPI、USB、CAN 等。其中 UART 串口（通用异步收发传输器（Universal Asynchronous Receiver/Transmitter））是一种常用的异步收发传输方式。

在常见通信总线协议中，I2C，SPI 属于同步通信而 UART 属于异步通信。同步通信的通信双方必须先建立同步，即双方的时钟要调整到同一个频率或边沿，收发双方不停地发送和接收连续的同步比特流。异步通信在发送字符时，发送端可以在任意时刻开始发送字符，所以，在 UART 通信中，数据起始位和停止位是必不可少的。从电气标准及协议来讲，串口包括了 RS-232、RS-422、RS485 等常见的通信方式。

2. MSPM0 UART 简介

作为嵌入式系统中使用频率最高的接口，UART 异步通信（也有同步应用的场景）的优点非常多，数据结构简单，只需要两根线进行收发，而且可以双工工作，波特率从低至 1200 bps 到常用的 115200 bps，以及更高到几 Mbps 波特率，在成本和性能上可以满足多种应用环境。MSPM0 在 UART 外设上做了深化改进，提供了更加丰富的功能组合，同时在低功耗方面也做了优化，以适应各种不同的应用场景。

2.1 MSPM0 UART 特点

- 支持 5、6、7 或 8bit 数据的接收和发送
- Even、odd、stick 或 no-parity 的校验方式
- 1 或 2bit 停止位
- LSB first 或 MSB first 可配置
- Line-Break 检测
- 输入端毛刺信号滤波
- 可编程的 3、8 或 16 倍过采样方式
- 深度为 4 的独立收发 FIFO
- 支持 DMA 数据收发
- 支持包括 stop、standby 模式的各种低功耗模式下正常工作
- 支持 loopback 模式，硬件流控
- 支持 9bit 以实现多机通信模式
- 并支持以下协议：
  - LIN 总线
  - DALI 总线
  - IrDA 红外收发
  - ISO7816 Smart Card 智能卡
  - RS485 总线
  - Manchester coding 曼彻斯特编码
  - Idle-Line Multiprocessor

Features	UART Extend	UART Main
Hardware flow control	Yes	Yes
Oversampling options	3, 8, 16	3, 8, 16
Separate transmit and receive FIFOs	Yes	Yes
Active in all low-power modes	Yes(2)	Yes (2)
Wake-up with start bit	Yes(3)	Yes(3)
Digital Glitch filter	Yes	No
Analog Glitch filter	Yes	Yes
9-bit multi-drop configuration	Yes	Yes
Idle-Line Multiprocessor	Yes	Yes
RS-485	Yes	Yes
Support LIN mode	Yes	-
Support DALI	Yes	-
Support IrDA	Yes	-
Support ISO7816 Smart card	Yes	-
Support Manchester code	Yes	-

(1) 参考具体 MCU 型号的数据手册以确定每个 UART 模块的功能以及所在的电源域；
(2) 除了 PD1 电源域的 UART 端口，其它 UART 都支持在 stop 和 standby 的低功耗模式下工作；
(3) 此功能仅适用于 PD0 电源域下面的 UART 模块；

图 1. UART 标准功能和扩展功能对比（UART Main and Extend Feature）

下图是 MSPM0 的 UART 模块的原理框图。

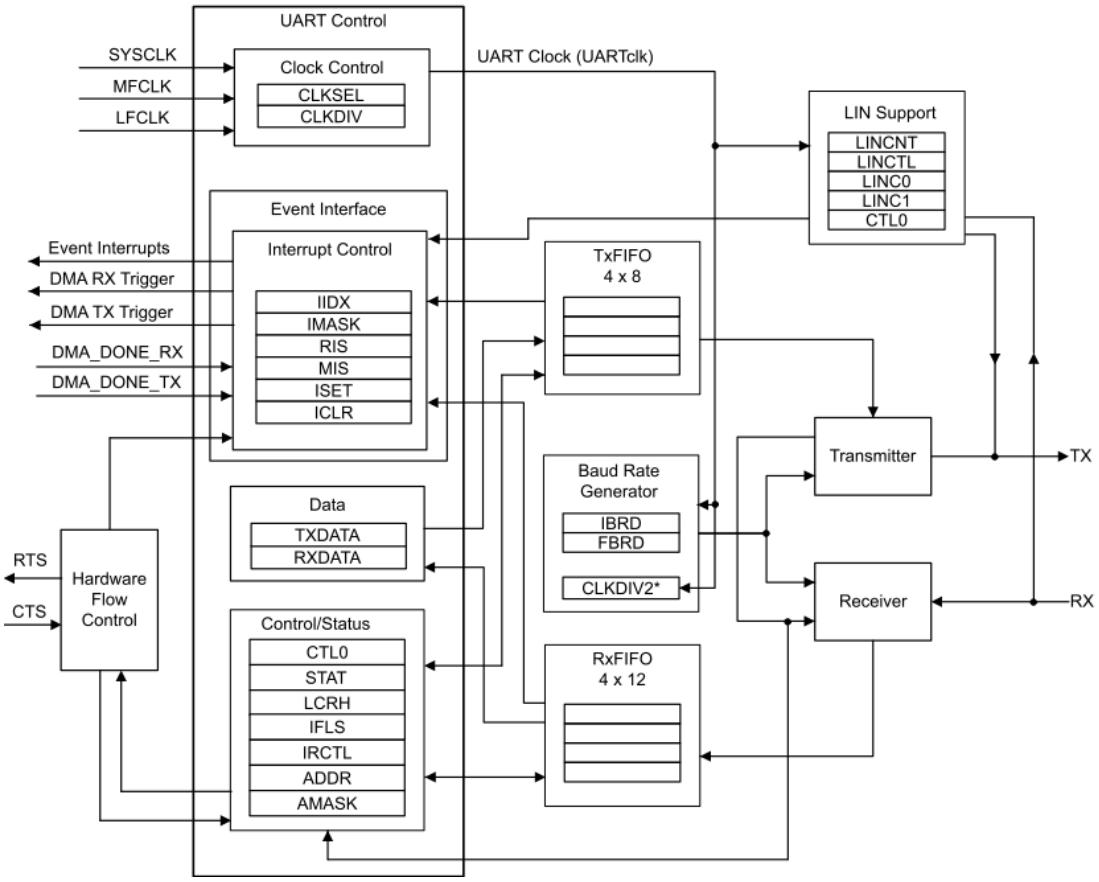

图 2. MSPM0 UART 模块原理框图

2.2 MSPM0 UART 外设功能介绍

2.2.1 UART 时钟及波特率

MSPM0 UART 模块的时钟源比较丰富，可以由 BUSCLK、MFCLK 或者 LFCLK 来提供，这意味着 UART 模块的最高时钟频率可以达到与 MCU 主频同频的 80MHz，这也是 MSPM0 UART 模块可以支持高速波特率的原因之一。在两个不同的 Power Domain 中，MSPM0G 系列 UART 的波特率最高能够达到 10Mbps，MSPM0L 系列的最高波特率为 4Mbps。

PARAMETERS		TEST CONDITIONS	MIN	TYP	MAX	UNIT
f_{UART}	UART input clock frequency				32	MHz
f_{BITCLK}	BITCLK clock frequency(equals baud rate in MBaud)				4	MHz
t_{SP}	Pulse duration of spikes suppressed by input filter	AGFSELx = 0	5	5.5	32	ns
		AGFSELx = 1	8	15	55	ns
		AGFSELx = 2	18	38	115	ns
		AGFSELx = 3	30	74	165	ns

图 3. MSPM0 UART 时钟及波特率

而 LFCLK 的频率为 32768Hz，这为低功耗情况低码率（<=9600bps）传输提供了可能。

2.2.2 UART 收发 FIFO

在 UART 模块的收发寄存器端，提供了深度为 4 的 FIFO 单元，收发的 FIFO 单独分开，可以通过 UART Data (TXDATA/RXDATA)寄存器进行访问。在读取 RXDATA 的时候会返回 12bit 的数据，包含了 8bit Data 和 4bit 的 error flag，写入 TXDAT 寄存器 8bit 数据会自动压进发送 FIFO。

在 MCU 上电或者复位后，所有的 FIFO 默认是禁止状态，这时 UART 的 FIFO 会作为 1Byte 的深度进行数据收发，需要手动配置 UARTx.CTL0 的 FEN 位使能 FIFO 功能。

FIFO 的状态有硬件实时监测，通过判断 FIFO 数据深度、empty、full 或者 overrun 等状态产生中断或事件，以方便用户进行相应的数据处理，这里 FIFO 的深度阈值可以被配置为 1/4、2/4、3/4 或者 full 状态。例如，如果配置串口接收 FIFO 的触发阈值被配置为 3/4 的 FIFO 深度，那么 UART 会在接收到 3 个字节的时候产生中断，通知 MCU 进行数据搬移或处理。

2.2.3 Glitch Suppression 毛刺抑制

MSPM0 的 UART 模块拥有两种 Glitch 消除的方式，Digital Filter 和 Analog Filter，它们相关的工作原理不同，处理的 Glitch 情况也不同。

Digital Filter：数字滤波器的工作是基于 UART 模块的时钟，通过 UARTx.GFCTL 寄存器的 DGFSEL 位来进行配置，在 RX 使能数字滤波功能后，所有接收到的信号都会根据配置进行一定 Clock 的延迟，然后再去进行数据采样，这个过程与 UART 的波特率紧密相关，最大的滤波延迟不能超过 1/3 波特率的脉冲宽度，以避免波特率误差过大。

Analog Filter：模拟滤波是通过内部集成的模拟滤波器来实现的，可以通过 UARTx.GFCTL 寄存器的 AGFSEL 位来进行配置，再通过 AGFEN 使能，不同的 MCU 型号在此参数上有区别，使用的时候需要注意。以 MSPM0G3507 为例，我们可以看到 AGFSELx = 3 的时候，其滤波能力可以滤除最大 165ns 时间宽度的噪声。

PARAMETERS		TEST CONDITIONS	MIN	TYP	MAX	UNIT
t_{SP}	Pulse duration of spikes suppressed by input filter	AGFSELx = 0	5	5.5	32	ns
		AGFSELx = 1	8	15	55	ns
		AGFSELx = 2	18	38	115	ns
		AGFSELx = 3	30	74	165	ns

图 4. MSPM0 UART 模拟滤波参数

2.3 UART 初始化流程

在 UART 模块配置之前，需要先清除 ENABLE 位，这样会彻底复位 UART 模块，以确保其状态处于一个正常可控的模式。如果要对 UART 模块进行初始化和使能，需要按照以下流程来操作。

- 配置相应的 GPIO 管脚，并通过 IOMUX 选择其为对应的 TXD 和 RXD 功能；
- 通过 UARTx.RSTCTL 寄存器复位 UART 模块；
- 通过 UARTx.PWREN 使能 UART 外设；
- 通过 UART.CLKSEL 和 UART.CLKDIV 寄存器配置 UART 模块的时钟源和分频系数；
- 清除 UART.CTL0.ENABLE 位以禁止 UART；
- 进行 UART 波特率配置，这里涉及到 UARTx.IBRD 和 UARTx.FBRD 寄存器；
- 配置计算出的 BRD 数值的整数部分到 UART.IBRD 寄存器；
- 配置计算出的 BRD 数值的小数部分到 UART.FBRD 寄存器；
- 通过 UART.CTL0 寄存器进行 FIFO 功能的配置；
- 将 UART 的其它参数写入 LCRH 寄存器，例如位长、STOP 位长度、奇偶校验、RS485 参数等；
- 通过 UART.CTL0.ENABLE 位使能 UART；

这样就完成了整个 UART 的配置流程，过程比较复杂，当然，用户可以利用 TI 提供的 Sysconfig 图形化界面工具非常方便的生成 UART 的初始化代码，相关详细信息可以通过以下链接访问：

https://www.ti.com.cn/tool/cn/SYSCONFIG

2.4 UART 中断和 Event 支持

MSPM0 最新集成的 Event Manager 功能提供了 peripheral-to-peripheral, peripheral-to-DMA, and peripheral-to-CPU (IRQ) 的 event 联通功能，各个 event 的发送端（publishers）和接收端（subscribers）通过一系列的静态路由和可编程路由通道进行连接。

不同于 MCU 中常见的系统中断事件，Event System 更加灵活和强大，除了支持通用的 IRQ 方式，还在不同外设之间提供了更加灵活的事件触发方式，甚至包含了在不同低功耗工作模式下的自动对 Power 和 Clock 的相应配置。

这些通过 Event Manager 进行传输的 Event 包含：

- 外设 Event 作为 IRQ 传输到 CPU（例如：RTC 中断传输到 CPU）
- 外设 Event 作为 DMA 触发事件传输到 DMA 模块（例如：UART 数据接收触发 DMA 的传输请求）

外设 Event 传输到另一个外设去直接触发这个外设的动作（例如：TIMx Timer 这个外设发送一个周期性的 Event 到 ADC 的事件接收端，然后 ADC 使用这个 Event 去周期性的触发其开始采样这个动作）

MSPM0 的 UART 模块包含了 3 个 event publishers，没有 event subscribers。

event publisher (INT_EVENT0) 通过静态路由负责 UART 到 CPU 的中断请求 IRQs，第二个和第三个 event publishers (INT_EVENT1, INT_EVENT2) 用来设置触发事件，以通过 DMA 事件路由进行 DMA 传输。下图是 UART Events 的概览。

Event	Type	Source	Destination	Route	Configuration	Functionality
CPU interrupt	Publisher	UART	CPU Subsystem	Static route	INT_EVENT0 registers	Fixed interrupt route from UART to CPU
DMA trigger	Publisher	UART	DMA	DMA event route	INT_EVENT1 registers	Fixed interrupt route from UART to DMA
DMA trigger	Publisher	UART	DMA	DMA event route	INT_EVENT2 registers	Fixed interrupt route from UART to DMA

图 5. MSPM0 UART 模块 Event 事件

2.5 UART 调试模式

在 MCU 应用中进行 UART 调试的时候，可能会遇到程序停在断点时，数据在后台依然会继续接收刷新寄存器，对调试产生了影响，这里我们可以通过 UART 模块的 PDBGCTL 寄存器中 FREE 和 SOFT 位来控制，禁止在断点中断后彻底停止 UART 模块的工作。

PDBGCTL.FREE	PDBGCTL.SOFT	Function
1	x	Modules continues operation
0	0	Module stops immediately
0	1	Module stops after the next transfer has been finished

图 6. MSPM0 UART 模块调试选项

这里我们可以看到 FREE 和 SOFT 配置为 0:0 的时候，模块在断点处立刻停止工作，方便用户查看当前寄存器的实时数值。

3.几种常规 UART 数据传输方式

在大多数应用中，我们利用 UART 两线方式进行数据传输，短距离的传输（一般为板对板或者板内传输）可以直接采用 TTL 电平方式，远距离的传输会采用 RS232 电平方式，其波特率覆盖 1200bps 到几 Mbps 级别，是一种灵活的传输方式。实际使用过程中，由于不同的系统应用中运行任务复杂程度不一，串口作为一个实时传输端口，面临着数据收发完整性和实时性的挑战，常规的 UART 数据传输方式一般分为：阻塞方式收发数据（实际应用中仅限于某些特殊场合）、UART 中断模式收发数据、UART 中断+FIFO 模式、UART+FIFO+DMA 模式，除了这些常规的模式，MSPM0 还提供了一种基于事件驱动的模式，我们可以称作 UART+FIFO+Event+DMA 模式，以下分别进行相关介绍。

3.1 UART 中断收发数据

UART 中断模式是一种常用模式，适用于非大批量数据的间断性收发数据，在这种机制下，每收到或发送一个字节，都会触发一个对应的收发中断，用户进入中断应用程序，对中断标志进行判断，接收或发送数据，同时相应的中断标志位会硬件自动清除。

这种模式面临着几种挑战：

1.如果数据收发较为密集，MCU 需要不停的响应中断，这样造成 MCU 负载过高；

2.由于是中断模式，如果应用中有多个高优先级的中断，则有可能会将 UART 中断挂起，产生数据丢失的风险；

3.2 UART 中断 + FIFO 收发数据

为了在中断模式的情况下减少 MCU 中断响应的频率，MSPM0 在 UART 模块端加入了 FIFO 单元，其深度为 4，发送和接收端独立分开，意味着可以对 4 个数据同时进行收发的缓冲，给串口收发带来了 4 倍时间的冗余度。在 FIFO 的加持下，如果还是利用中断模式收发数据，那么 MCU 的中断响应次数可以减少到之前的 1/4。

MSPM0 的 FIFO 单元提供了硬件实时监测功能，通过判断 FIFO 数据深度、empty、full 或者 overrun 等状态产生中断或事件，以方便用户进行相应的数据处理，这里 FIFO 的深度阈值可以被配置为 1/4、2/4、3/4 或者 full 状态。例如，如果配置串口接收 FIFO 的触发阈值被配置为 3/4 的 FIFO 深度，那么 UART 会在接收到 3 个字节的时候产生中断，通知 MCU 进行数据搬移或处理，一般情况下，用户不会等到 FIFO 满才去处理数据。

3.3 UART + DMA 收发数据

由于 FIFO 的长度是有限的，所以对于长包数据的传输，或者频繁的数据收发，FIFO 能够缓解的效果有限，在利用 DMA 通道进行数据搬移的情况下，可以大大的解放 MCU 的负担。DMA 通道的工作模式比较简单，在配置了 DMA 的数据源地址、目的地址和搬移数据长度后，确定 DMA 开始传输的触发源即可让 DMA 自动工作。例如针对 UART 接口，用户可以利用 MSPM0 的 UART 的 TXINT、RXINT、RTOUT 事件来触发 DMA 开始传输，将数据搬移到指定的 RAM 地址。

其中接收数据这里分两种不同的情况来触发：

- **FIFO 使能的情况下：**如果接收到的数据达到了 FIFO 预置的触发阈值，那么 RXINT 位会置 1，DMA 在读取 FIFO 数据后，RXINT 会自动清零。如果对 IIDX（Interrupt index）进行读操作或者对 ICLR 寄存器中 RXINT 位进行写 1 操作，也会清除 RXINT 中断标志位。
- **FIFO 未使能的情况下：**如果没有使能 FIFO，那么每接收到一个数据，都会触发 RXINT 置 1，同时 DMA 开始搬移数据，这时 RXINT 自动清零。如果对 IIDX（Interrupt index）进行读操作或者对 ICLR 寄存器中 RXINT 位进行写 1 操作，也会清除 RXINT 中断标志位。

在这种模式下，可以通过 DMA 进行大批量长数据的搬移，无论是连续数据，还是零散数据，都可以很好的保持数据完整性，是大部分应用中效率最高、最常用的模式。

3.4 MSPM0 UART + FIFO + DMA 收发数据

由于 MSPM0 的串口提供了收发 FIFO，所以即使在有 DMA 的情况下，也建议客户把 FIFO 使用起来，因为 DMA 通道的数据传输会占用 MCU 总线时间片，多个 DMA 通道的传输和 MCU 总线会有冲突的情况发生，DMA 通道之间的仲裁机制是有自动优先级来分配资源，总线的仲裁也是自动分配，当然这个时间非常短，

跟总线频率相关，合理的程序设计完全可以保证数据的完整性，即便如此，FIFO 的使用会大大放宽对总线带宽的要求。

4. 环形缓冲 Ring Buffer 在串口上的应用

缓冲区 Buffer 是在收发数据时经常用到的功能，从字面意思看，其功能就是缓冲数据用的。实现缓冲区最简单的办法是采用数组的方式在 RAM 中开辟一定长度的空间，将接收到的数据存入其中，当然也可以利用 Ping-Pong 模式定义两个缓冲区 Buffer，当 Buffer1 满的时候切换到 Buffer2，继续收发数据，同时可以对 Buffer1 的数据进行处理，这种方式基本可以满足普通应用，但是有一定的缺点。

- 缓冲区 Buffer 数组的长度是固定的，在应用过程中需要用户对代码中用到的最长缓冲区长度作为数组的长度，这将会导致 RAM 占用量增加，如果采用 Ping-Pong 模式定义两个缓冲区 Buffer，那么 RAM 占用量会更加明显，对于低成本 MCU 来讲是一大挑战。
- 实际应用过程中，接收的数据长度有可能无法预计到最长的接收长度，那可能意味着定义的数组无法满足数据存储，造成溢出。
- 数组的 RAM 利用效率低下，除了数组的长度过长外，还有 Buffer2 处于待机状态，没有充分利用 Buffer1 缓冲节省出来的 MCU 时间片。

而环形缓冲 Ring Buffer 可以很好的解决这些问题，使用简单，拥有更高的 RAM 利用率。

4.1 环形缓冲 Ring Buffer 简介

下面是环形缓冲 Ring Buffer 的结构示意图，其本质是一个环形队列结构，一个带“头指针（指向读）”和“尾指针（指向写）”的数组。

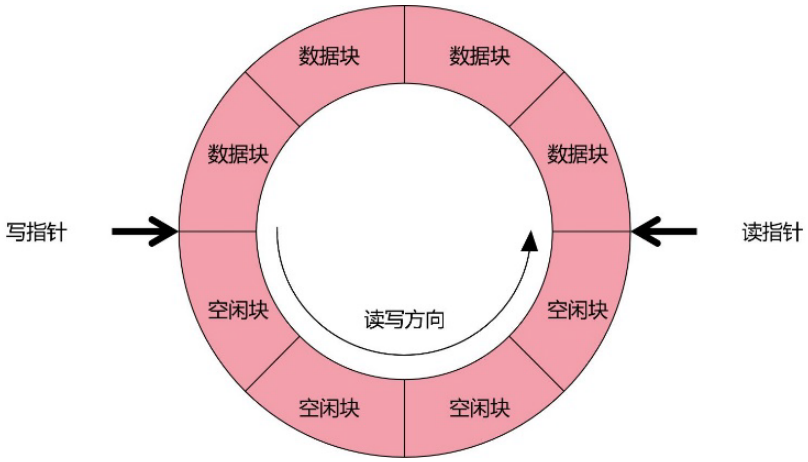

图 7. Ring Buffer 环形缓冲示意图

一般情况下，圆形缓冲区需要 4 个指针：

1. 在内存中实际开始位置，即 Ring Buffer 数组的地址；
2. 在内存中实际结束位置，也可以用缓冲区长度代替，即 Ring Buffer 的长度；
3. 存储在缓冲区中的有效数据的开始位置（读指针）；
4. 存储在缓冲区中的有效数据的结尾位置（写指针）。

“头指针”指向环形缓冲区中可读的数据，“尾指针”指向环形缓冲区中可写的缓冲空间。通过移动“头指针”和“尾指针”就可以实现缓冲区的数据读取和写入。在通常情况下，应用程序读取环形缓冲区的数据仅仅会影响“头指针”，而串口接收数据仅仅会影响“尾指针”。当串口接收到新的数组，则将数组保存到环形缓冲区中，同时将“尾指针”加 1，以保存下一个数据；应用程序在读取数据时，“头指针”加 1，以读取下一个数据。当“尾指针”超过数组大小，则“尾指针”重新指向数组的首元素，从而形成“环形缓冲区”，有效数据区域在“头指针”和“尾指针”之间。

4.2 UART 串口通信中环形缓冲 Ring Buffer 的应用

这里可以定义一个 Ring Buffer 的结构体如下。

```
typedef struct {
    unsigned char      *buffer;
    size_t              length;
    size_t              count;
    size_t              head;
    size_t              tail;
    size_t              maxCount;
} RingBuf_Object, *RingBuf_Handle;
```

buffer 为定义的环形缓冲区在 RAM 中的起始地址，length 为缓冲区长度，count 为当前 buffer 中存储的数据个数，head 为缓冲区的头，tail 为缓冲区的尾，maxCount 为缓冲区的最大使用长度（用作性能评估，检查余量）。

这里我们几个常用的针对 Ring Buffer 操作的 API 函数分别包含：

- Ring Buffer的创建：`void RingBuf_construct(RingBuf_Handle object, unsigned char *bufPtr, size_t bufSize)`
- Ring Buffer读取单个数据：`int RingBuf_get(RingBuf_Handle object, unsigned char *data)`
- Ring Buffer读取缓冲区内数据个数：`int RingBuf_getCount(RingBuf_Handle object)`
- Ring Buffer读取n个数据：`int RingBuf_getn(RingBuf_Handle object, unsigned char *data, size_t n)`
- Ring Buffer里写单个数据：`int RingBuf_put(RingBuf_Handle object, unsigned char data)`
- Ring Buffer里写n个数据：`int RingBuf_putn(RingBuf_Handle object, unsigned char *data, size_t n)`

创建 RingBuf：

```
void RingBuf_construct(RingBuf_Handle object, unsigned char *bufPtr,
    size_t bufSize)
{
    object->buffer = bufPtr;
    object->length = bufSize;
    object->count = 0;
    object->head = bufSize - 1;
    object->tail = 0;
    object->maxCount = 0;
}
```

这里初始化了 RingBuf 的 RAM 空间指针，长度，当前 count 值，head 的下标，tail 的下表，以及 RingBuf 的最大使用长度。

RingBuf 单个数据读取：

```
int RingBuf_get(RingBuf_Handle object, unsigned char *data)
{
    gINT_disable();

    if (!object->count) {
        gINT_enable();
        return -1;
    }

    *data = object->buffer[object->tail];
    object->tail = (object->tail + 1) % object->length;
    object->count--;

    gINT_enable();

    return (object->count);
}
```

这里需要注意的是，在进行 RingBuf 读取之前，为了防止环形缓冲区有被中断打断，在中断应用程序中操作的可能性，需要临时关闭中断，操作完成后，重新打开中断，返回剩余数据的个数。

```
int RingBuf_getCount(RingBuf_Handle object)
{
    return (object->count);
}
```

RingBuf_getCount 用来获取当前缓冲区内数据个数的信息。

```
int RingBuf_getn(RingBuf_Handle object, unsigned char *data, size_t n)
{
    size_t       removed = 0;

    gINT_disable();
    if (n > object->count) {
        n = object->count;
    }
    while (n) {
        *data++ = object->buffer[object->tail++];
        object->tail %= object->length;
        --object->count;
        --n;
        ++removed;
    }
    gINT_enable();

    return (removed);
}
```

RingBuf_getn 可以一次获取多个数据，可以看到函数内会进行 n 与当前剩余数据的对比，防止出错，最终返回取数的个数。

```
int RingBuf_put(RingBuf_Handle object, unsigned char data)
{
    unsigned int next;

    gINT_disable();

    if (object->count != object->length) {
        next = (object->head + 1) % object->length;
        object->buffer[next] = data;
        object->head = next;
        object->count++;
        object->maxCount = (object->count > object->maxCount) ?
                           object->count :
                           object->maxCount;
    }
    else {

        gINT_enable();
        return (-1);
    }

    gINT_enable();

    return (object->count);
}
```

RingBuf_put 可以实现单个数据的写入，并返回当前缓冲区的长度。

```
int RingBuf_putn(RingBuf_Handle object, unsigned char *data, size_t n)
{
    size_t       next;
    size_t       added = 0;

    gINT_disable();

    if (n > RingBuf_space(object)) {
        n = RingBuf_space(object);
    }
    while (n) {
        next = (object->head + 1) % object->length;
        object->buffer[next] = *data++;
        object->head = next;
        ++object->count;
        --n;
        ++added;
    }
    if (object->maxCount < object->count) {
        object->maxCount = object->count;
    }

    gINT_enable();

    return (added);
}
```

RingBuf_putn 实现了多个数据的一次性写入，并返回写入的个数已方便校验。

下面搭建一个简单的 RingBuf 测试应用，在 main 函数中先通过 RingBuf_construct 初始化一个 pUartRingBufHandle 的结构体，其缓冲区指向 gUartRingBuf 这个 RAM 地址，长度为 256 字节。

```
int main(void)
{
  unsigned char i;
  unsigned char cnt;
  unsigned char t[256];

  SYSCFG_DL_init();

  NVIC_ClearPendingIRQ(UART_0_INST_INT_IRQN);
  NVIC_EnableIRQ(UART_0_INST_INT_IRQN);

  RingBuf_construct(&pUartRingBufHandle, gUartRingBuf, 256);

  while (1)
  {
    cnt = RingBuf_getCount(&pUartRingBufHandle);

    if(cnt !=0)
    {
      RingBuf_getn(&pUartRingBufHandle, t, cnt);

      for(i=0;i<cnt;i++)
        DL_UART_transmitDataBlocking(UART_0_INST, t[i]);
    }
  }
}
```

在 while 循环里面可以进行当前环形缓冲区存储数据个数的查询，只要有数据在里面，就可以通过 RingBuf_getn 读取数据，这里我们通过 DL_UART_transmitDataBlocking 函数用串口打印到终端进行查看校验。

RingBuf 缓冲区的数据是串口接收到的数据，我们可以在串口的接收中断里进行填充，如下图代码。

```
void UART_0_INST_IRQHandler(void)
{
  switch (DL_UART_Main_getPendingInterrupt(UART_0_INST)) {
      case DL_UART_MAIN_IIDX_RX:
          data = DL_UART_Main_receiveData(UART_0_INST);

          RingBuf_put(&pUartRingBufHandle, data);
          break;
      default:
          break;
  }
}
```

在串口中断内，判断为接收中断后，将接收到的数据（1 个字节）从 RX Buffer 中读取，并通过 RingBuf_put 写入循环缓冲区。

此测试简单的完成了 RingBuf 的创建、写入、读取等应用，在实际测试中，当串口的波特率达到 921600 的时候，每包发送 128 字节，都可以正常接收并在 while 循环中打印输出，无误码出现，而按照传统的中断接收，并在中断中 echo 的方式，波特率在 256000 的时候就产生了误码丢数据，原因在于串口发送的等待时间较长且不可控，当然也可以采取中断发送的方式，但这样会增加串口中断的频率。

上面这个简单的例程只是为了快速了解 RingBuf 环形缓冲区带来的有点，在实际应用中并不一定合适，因为 UART 模块的 FIFO 以及 DMA 功能并没有使用起来，这样会大大减少 MCU 的中断频率，下一节会针对这个问题进行详细讨论。

5. 使用 DMA + Ring Buffer 提高串口吞吐率和灵活性

为了提高串口数据的吞吐率，MSPM0 在模块中增加了深度为 4 的收发 FIFO，而且可以利用 DMA 通道进行数据的收发搬移，将这些功能结合起来，再加上 Ring Buffer 的使用，会使得串口吞吐率更高，CPU 资源占用率更低，功耗更低，而且会更加灵活的处理不同的数据格式。

5.1 使用 FIFO 和 DMA 进行串口通信

这里我们先进行 FIFO 操作的配置，在 sysconfig 中，使能 FIFO 功能，并配置收发 FIFO 的触发阈值为 2 字节，也就是 FIFO 半满的时候产生中断。

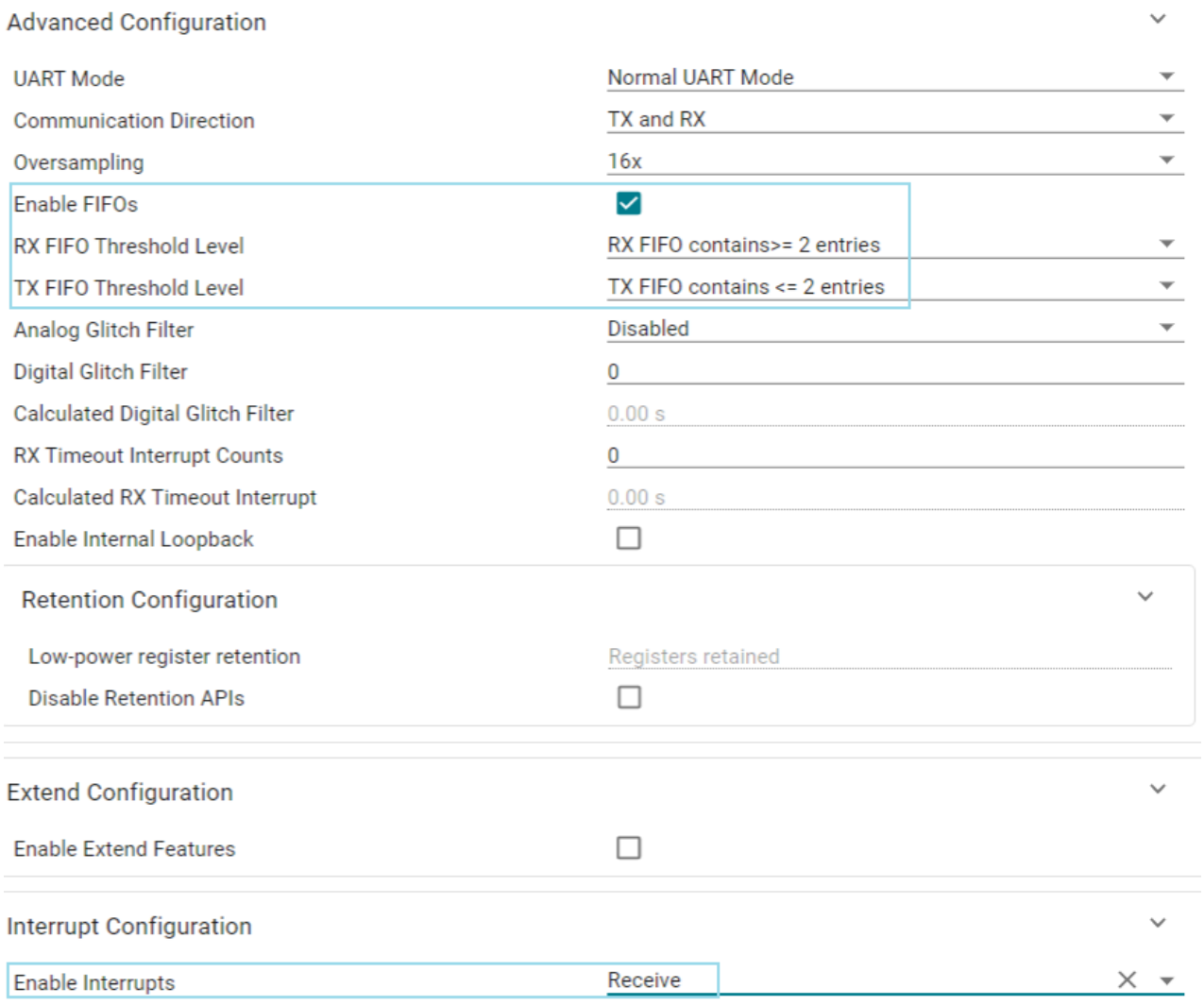

图 8. MSPM0 UART FIFO 配置

接下来使能 UART 的 Receive 中断，这个中断会在 FIFO 中的数据接收达到 2 字节的时候触发，以通知 MCU 进行读取数据操作，在中断应用程序里，我们可以通过判断 FIFO 是否为空来读取 FIFO 中所有的数据。

为了使能 DMA 功能，需要进行地址模式、DMA 搬移数据的格式、长度、传输模式等进行配置，DMA 的触发源可以选择为 UART RX 中断源，这里由于使能了 FIFO，所以 RX 的中断触发条件为 FIFO 阈值达到配置值来触发，下图为 DMA 相关配置。

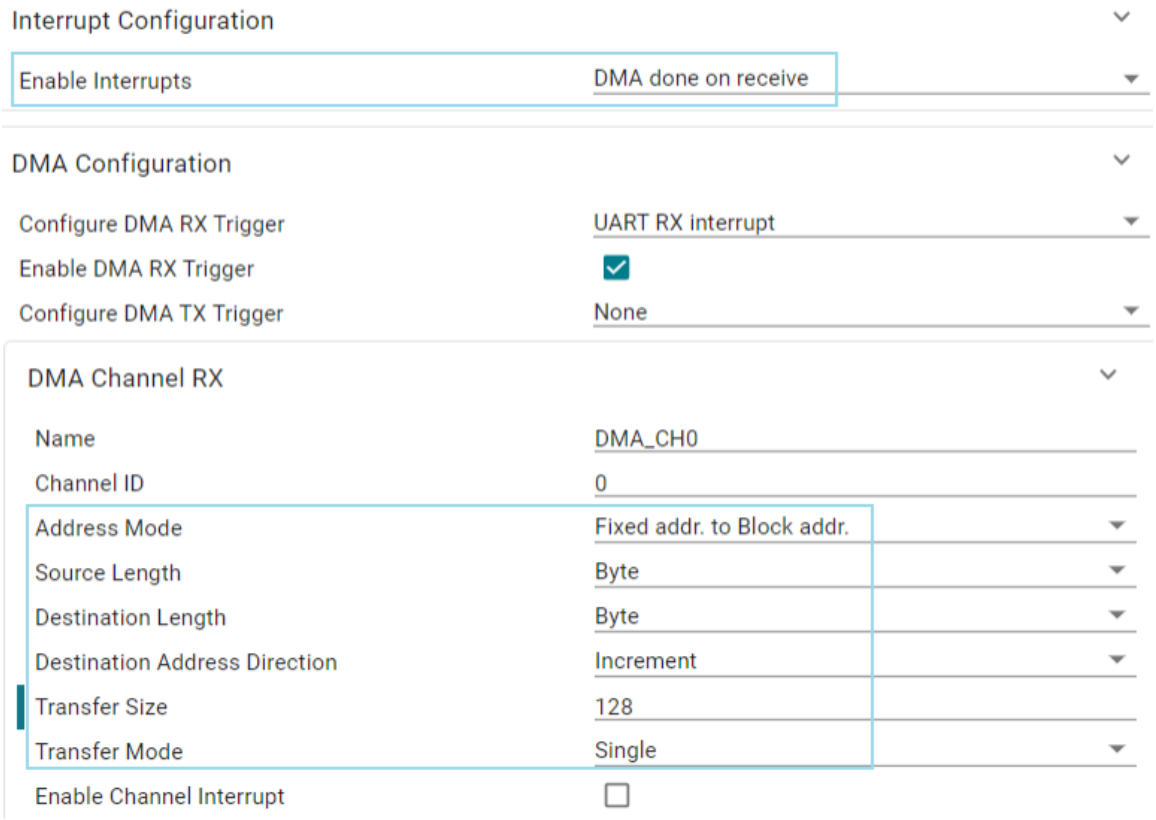

图 9. MSPM0 UART DMA 配置

在程序中配置 DMA 的参数如下，其源地址为 UART 的 RXDATA，由于已经启用了 FIFO，所以这里对应的也是 FIFO 的地址，目标地址为 gRxPacket 作为 DMA 缓冲 Buffer，TransferSize 根据实际应用的具体需求来进行定义，并使能 DMA 通道。

```
SYSCFG_DL_init();

NVIC_ClearPendingIRQ(UART_0_INST_INT_IRQN);
NVIC_EnableIRQ(UART_0_INST_INT_IRQN);

RingBuf_construct(&pUartRingBufHandle, gUartRingBuf, 256);

DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(&UART_0_INST->RXDATA));
DL_DMA_setDestAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t) &gRxPacket0[0]);
DL_DMA_setTransferSize(DMA, DMA_CH0_CHAN_ID, 64);
DL_DMA_enableChannel(DMA, DMA_CH0_CHAN_ID);

/* Confirm DMA channel is enabled */
while (false == DL_DMA_isChannelEnabled(DMA, DMA_CH0_CHAN_ID)) {
    __BKPT(0);
}
```

这里 DMA 的中断属于 UART 模块的一部分，所以在串口的中断应用程序中，读取 UART 的中断标志位进行判断，这里 DMA 对应的是 DL_UART_MAIN_IIDX_DMA_DONE_RX 中断标志位，代表 DMA 传输完成，用户可以在这里进行 DMA Buffer 切换和 RingBuffer 写入。

```
void UART_0_INST_IRQHandler(void)
{
  switch (DL_UART_Main_getPendingInterrupt(UART_0_INST)) {
      case DL_UART_MAIN_IIDX_RX:

          while(DL_UART_isRXFIFOEmpty(UART_0_INST)==false)
            RingBuf_put(&pUartRingBufHandle, DL_UART_receiveData(UART_0_INST));

          break;

      case DL_UART_MAIN_IIDX_DMA_DONE_RX:

        if(gDmaBufCnt == 0)
        {
          DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(&UART_0_INST->RXDATA));
          DL_DMA_setDestAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t) &gRxPacket1[0]);
          DL_DMA_setTransferSize(DMA, DMA_CH0_CHAN_ID, 64);
          DL_DMA_enableChannel(DMA, DMA_CH0_CHAN_ID);

          RingBuf_putn(&pUartRingBufHandle, gRxPacket0, 64);

          gDmaBufCnt = 1;
        }
        else
        {
          DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(&UART_0_INST->RXDATA));
          DL_DMA_setDestAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t) &gRxPacket0[0]);
          DL_DMA_setTransferSize(DMA, DMA_CH0_CHAN_ID, 64);
          DL_DMA_enableChannel(DMA, DMA_CH0_CHAN_ID);

          RingBuf_putn(&pUartRingBufHandle, gRxPacket1, 64);

          gDmaBufCnt = 0;
        }
        break;

      default:
          break;
  }
}
```

这里我们可以看到，DMA 的作用类似将 UART 自带的 FIFO 长度进行了扩展，使得原有的深度为 4 的 FIFO 变成了 64 或更大的深度，而且带来的 MCU 开销非常小，大大提高了高速通信过程中的缓冲能力，当然，这里有些缺点存在，比如 DMA 搬移的长度为固定的 64 或其它数值，在一些不固定长度的通信协议里就不够灵活友好，下面我们可以通过以下方式来改进。

5.2 更改 DMA 触发方式并加入 Ring Buffer 机制以增强数据搬移灵活性和吞吐率

为了让传输方式更加灵活，我们需要对 DMA 的触发方式进行分析，当前的 DMA 传输长度是固定的，如何做到长度可变，我们可以利用 UART 的接收超时机制在 FIFO 即使没有触发的情况下进行中断，这样可以读取 FIFO 剩余数据，同时需要开启 UART 的超时中断，作为判断 DMA 传输的截止条件，此时 DMA 需要配

置为 Repeat Single 模式，传输长度可配置为一个相当长的数据，这里不会作为 DMA 停止传输的条件，详细信息如下图所示。

Table 14-44. IFLS Register Field Descriptions

Bit	Field	Type	Reset	Description
31-12	RESERVED	R/W	0h	
11-8	RXTOSEL	R/W	0h	UART Receive Interrupt Timeout Select. When receiving no start edge for an additional character within the set bittimes a RX interrupt is set even if the FIFO level is not reached. A value of 0 disables this function. 0h = Smallest value Fh = Highest possible value

图 10. MSPM0 UART 模块接收超时寄存器配置

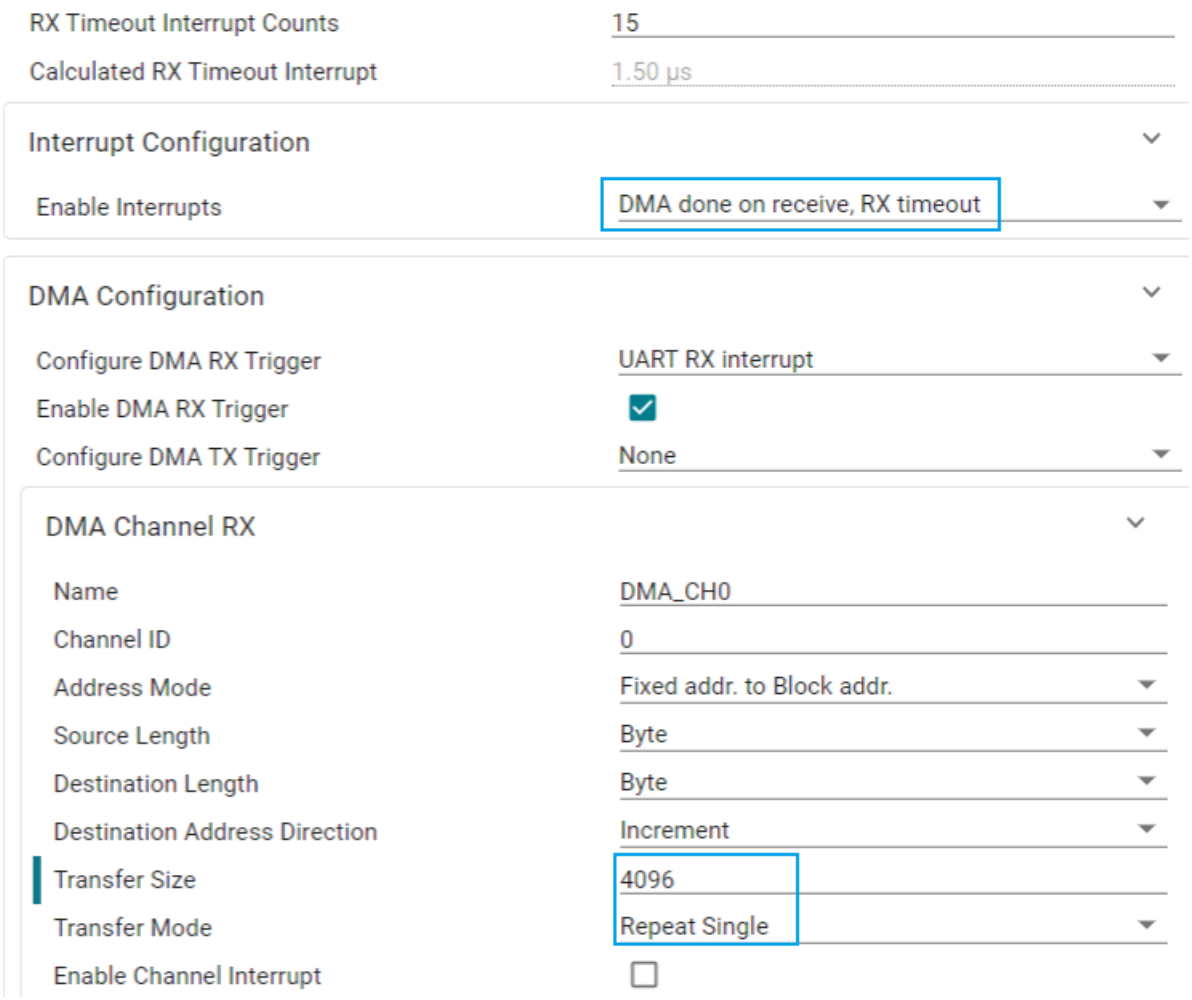

图 11. MSPM0 UART DMA 触发方式配置

在这种情况下，我们可以很方便的在不定长度的数据接收过程中尽可能的减少 MCU 中断，同时利用 Ring Buffer 环形缓冲区进行大批量数据的缓冲，在高通信速率情况下很有优势，避免数据的丢失，提高 CPU 的利用率。在实际测试中，当串口的波特率达到 4Mbps（16 倍采样方式）的时候，利用 DMA 和 Ring Buffer

模式，每包发送 4096 字节，都可以在另一块 MSPM0 板子上进行无误码接收（PC 端串口软件已无法支持如此高波特率），这种传输方式已应用在用户 PLC 扩展模块背板数据通信传输场合。

6 总结

在串口通信的各种应用场景中，不定长的数据帧，不同的波特率，用户自定义的数据包格式，以及单 MCU 的多串口同时收发，这些都给应用中带来了不同的挑战。灵活的运用 MSPM0 UART 模块中的 FIFO、DMA、Event 事件管理系统，并在应用中引入 Ring Buffer 循环缓冲机制，可以更加灵活的应对这些复杂的应用场景，在尽可能减少 CPU 中断频率的同时，减轻 CPU 负担，降低中断应用程序的执行时间，给 CPU 留有更多的资源进行数据包处理，以及其它外设操作和算法的处理，同时降低系统功耗，是一种高效而简易可行的方案。

7 参考文献

1. MSPM0G350x Mixed-Signal Microcontrollers Datasheet（SLASF83）
2. How Arm Cortex-M0+ MCUs optimize general-purpose processing, sensing and control
3. MSPM0 G 系列 MCU 硬件开发指南 (Rev. A)（SLAAE76A）
4. MSPM0 MCU 快速参考指南 (Rev. A)（SLAAE70A）
5. SysConfig 如何助推嵌入式系统开发

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265