SDK应用_DMA 模块配置及应用

版本：

Config Tool Version：2.0.5

YTM32B1ME0 SDK version：1.1.1

前言：

DMA 全称Direct Memory Access，是一个可以自主进行数据搬运的外设，可以有效的降低CPU 的负载，让CPU 有更多的时间进行数据计算和流程控制，避免因为数据搬运请求打断CPU 的执行过程。在软件设计中，合理的使用DMA 可以有效的提高系统总线利用率，外设模块也可以实现更高速率的数据收发。

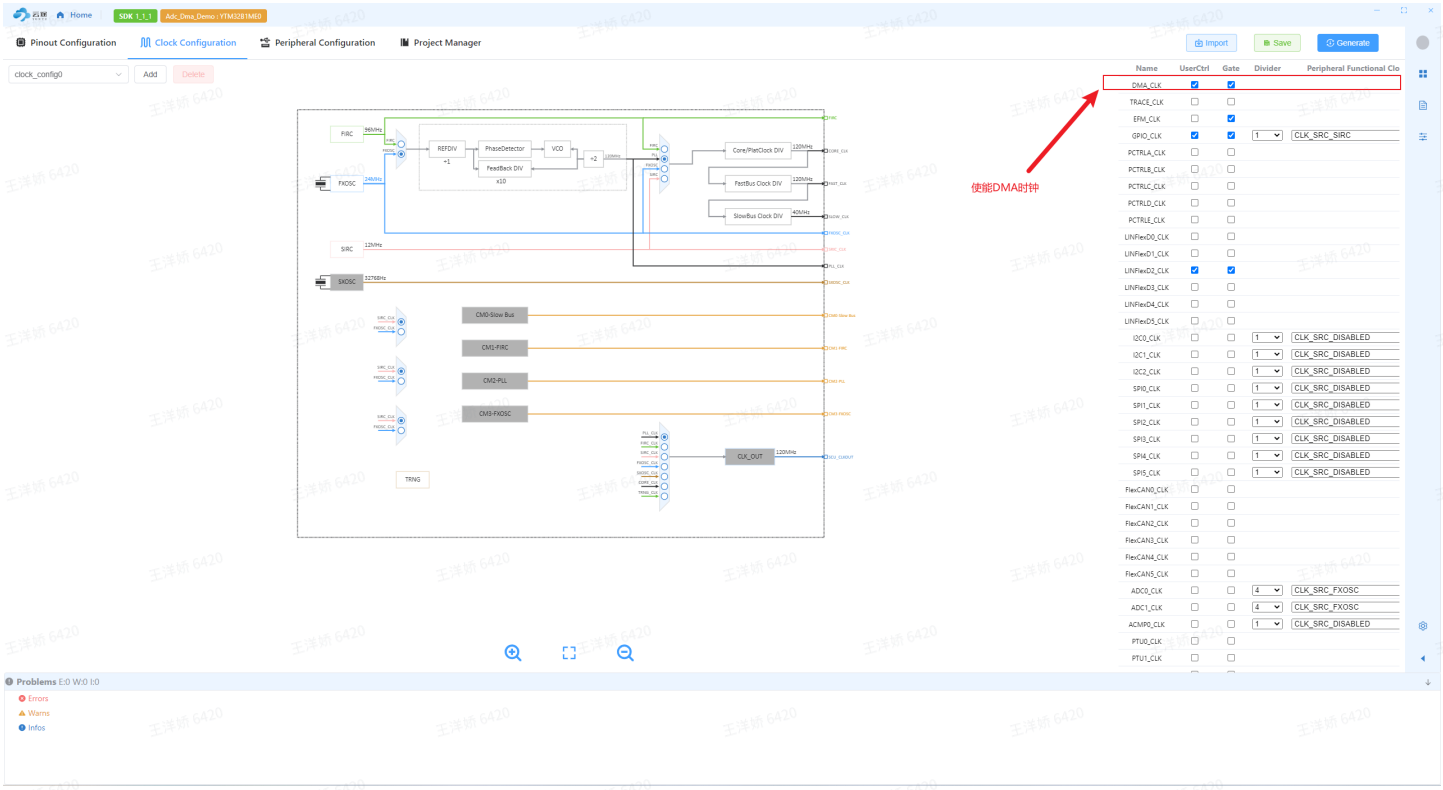
不同系列的DMA有些许不同，本文主要介绍 YTM32B1ME0 的 DMA 模块与配置。

关于 DMA 的应用笔记可参考 **AN_0030_DMA_ApplicationNote.pdf**

1. DMA in YCT 配置介绍

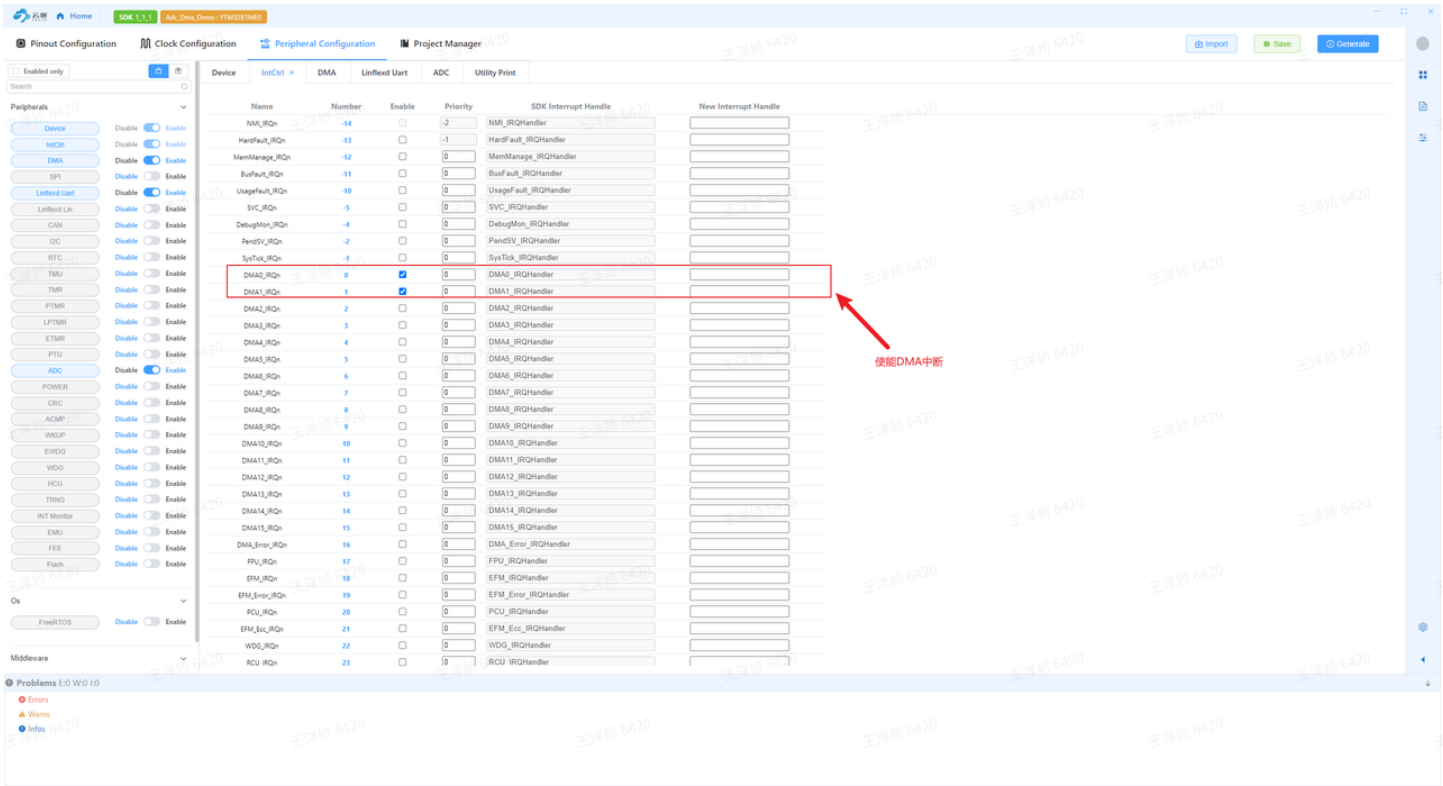
1.1 时钟配置

DMA 无法选择功能时钟，只需要使能该模块的功能时钟即可，系统时钟树用默认的就可以，如图是 120MHz PLL 作系统时钟，PLL 时钟源为 24MHz 外部晶振。



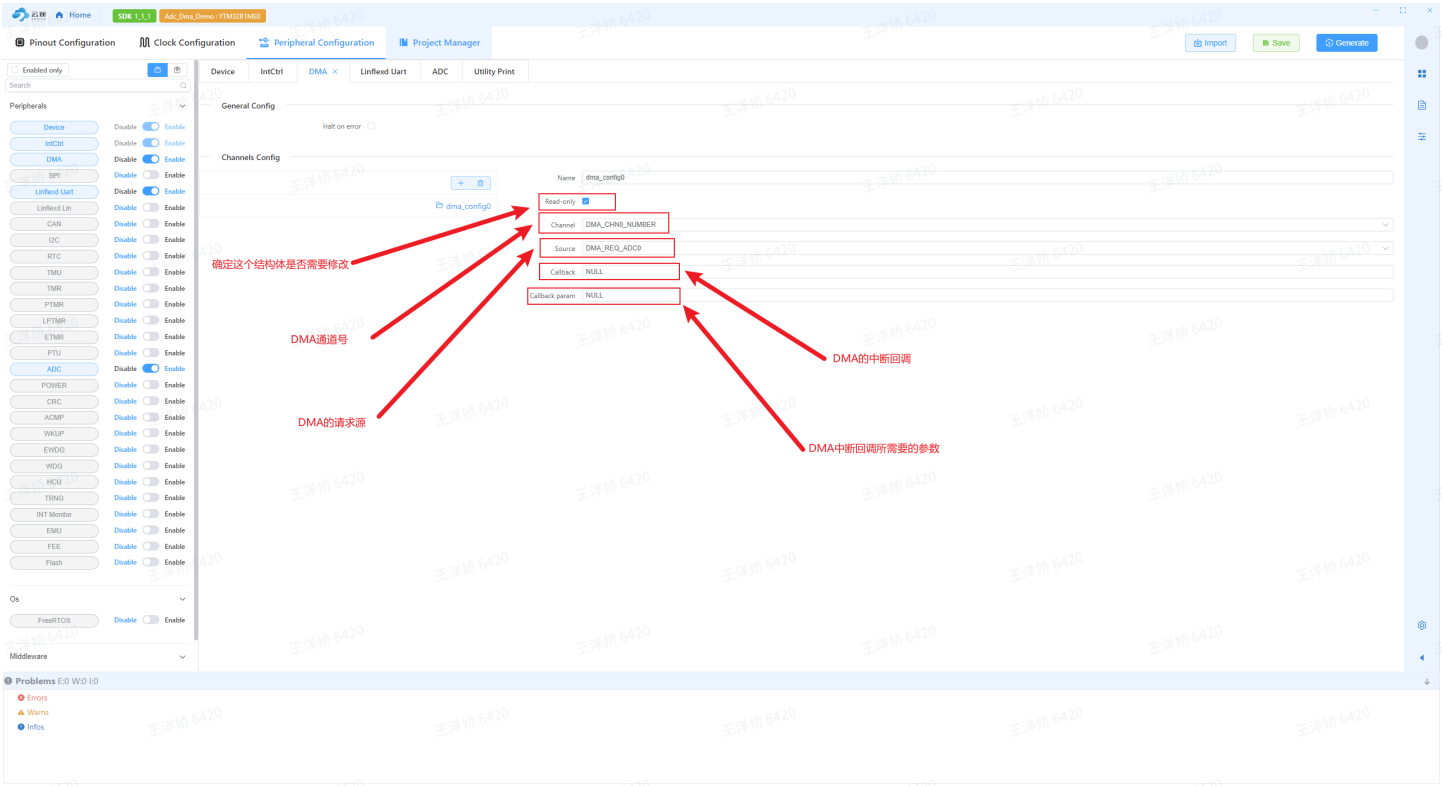
1.2 中断配置

使能需要用到的DMA中断



1.3 外设配置

外设配置解释如下



关于 YT Config Tool 的 DMA 模块相关配置主要就这些，其他外设与 DMA 相关的传输参数需单独配置。

2. DMA in SDK 配置介绍

2.1 DMA 基础术语

为了方便对DMA 特性的描述，约定如下的基础术语：

Trigger

触发事件，DMA 可以对外部模块或者软件进行响应，外部模块通过Trigger 事件请求DMA 进行数据传输

Access Loop

一次数据访问的byte 数量，单纯指一次总线访问，和DMA 的SSIZE，DSIZE 设置相关

Transfer Loop

DMA 一个通道一次触发需要搬运的bytes 数目，一次Transfer Loop 可能包含多个Access Loop

Trigger Loop

触发循环，代表DMA 的一个通道总共需要响应多少次Trigger 事件，每一次Trigger DMA 会搬运Transfer Loop 中定义的Bytes 数目的数据

CTS

Channel Transfer Structure，通道传输的配置结构体，CTS 中定义了DMA 通道数据传输的地址、偏移、循环

次数和触发配置等信息，DMA 通道基于CTS 信息决定对于DMA 通道触发信号的响应

2.2 DMA 传输配置结构体介绍

```
/*!
 * @brief DMA transfer size configuration.
 *
 * This structure configures the basic source/destination transfer attribute.
 * Implements : dma_transfer_config_t_Class
 */
typedef struct
{
    uint32_t srcAddr;                       /*!< Memory address pointing to the source data. */
    uint32_t destAddr;                      /*!< Memory address pointing to the destination data. */
    dma_transfer_size_t srcTransferSize;    /*!< Source data transfer size. */
    dma_transfer_size_t destTransferSize;   /*!< Destination data transfer size. */
    int16_t srcOffset;                      /*!< Sign-extended offset applied to the current source address
                                                 to form the next-state value as each source read/write
                                                 is completed. */
    int16_t destOffset;                     /*!< Sign-extended offset applied to the current destination
                                                 address to form the next-state value as each source
                                                 read/write is completed. */
    int32_t srcLastAddrAdjust;              /*!< Last source address adjustment. */
    int32_t destLastAddrAdjust;             /*!< Last destination address adjustment. Note here it is only
                                                 valid when ram reload feature is not enabled. */
    dma_modulo_t srcModulo;                 /*!< Source address modulo. */
    dma_modulo_t destModulo;                /*!< Destination address modulo. */
    uint32_t transferLoopByteCount;         /*!< Number of bytes to be transferred in each service request
```

```
                                                of the channel. */
    bool ramReloadEnable;                      /*!< Enable ram
reload feature. */
    uint32_t ramReloadNextDescAddr;            /*!< The address of
the next descriptor to be used, when
                                                ram reload feature
is enabled.
                                                Note: this value
is not used when ram reload
                                                feature is
disabled. */
    bool interruptEnable;                      /*!< Enable the
interrupt request when the trigger loop
                                                count completes */
    dma_loop_transfer_config_t *loopTransferConfig;     /*!< Pointer to loop
transfer configuration structure
                                                (defines
transfer/trigger loop attributes)
                                                Note: this field
is only used when transfer loop mapping is
                                                enabled from DMA
configuration. */
} dma_transfer_config_t;
```

上图为 DMA 的传输参数结构体

srcAddr

DMA搬运的源地址

destAddr

DMA搬运的目的地址

srcTransferSize

DMA搬运源地址的数据时的最小搬运单位，可选配置如下，若选择**DMA_TRANSFER_SIZE_4B**，即一次搬运4个bytes

```
/*! @brief DMA transfer configuration
 * Note: in L/Z series DMA_TRANSFER_SIZE_16B and DMA_TRANSFER_SIZE_32B are
not supported.
 * Implements : dma_transfer_size_t_Class
 */
typedef enum
{
```

```
    DMA_TRANSFER_SIZE_1B = 0x0U,
    DMA_TRANSFER_SIZE_2B = 0x1U,
    DMA_TRANSFER_SIZE_4B = 0x2U,
    DMA_TRANSFER_SIZE_8B = 0x3U,
    DMA_TRANSFER_SIZE_16B = 0x4U,
    DMA_TRANSFER_SIZE_32B = 0x5U,
    DMA_TRANSFER_SIZE_64B = 0x6U,
} dma_transfer_size_t;
```

destTransferSize

DMA搬运至目的地址的数据时的最小搬运单位，对应上文的 Access Loop，若选择 *DMA_TRANSFER_SIZE_4B*，即一次搬运4个bytes

srcOffset

源地址偏移，该参数可设置为负数。执行一次 Access Loop 后，

a. 用户希望依旧搬运这个地址的数据，可将 srcOffset 设置为 0

b. 用户希望搬运4个字节后的数据，可将 srcOffset 设置为 4

c. 用户希望搬运4个字节前的数据，可将 srcOffset 设置为 -4

destOffset

目的地址偏移，该参数可设置为负数。执行一次 Access Loop 后，

a. 用户希望依旧搬运数据至该地址，可将 destOffset 设置为 0

b. 用户希望搬运数据至该地址后4个字节的位置，可将 destOffset 设置为 4

c. 用户希望搬运数据至该地址前4个字节的位置，可将 destOffset 设置为 -4

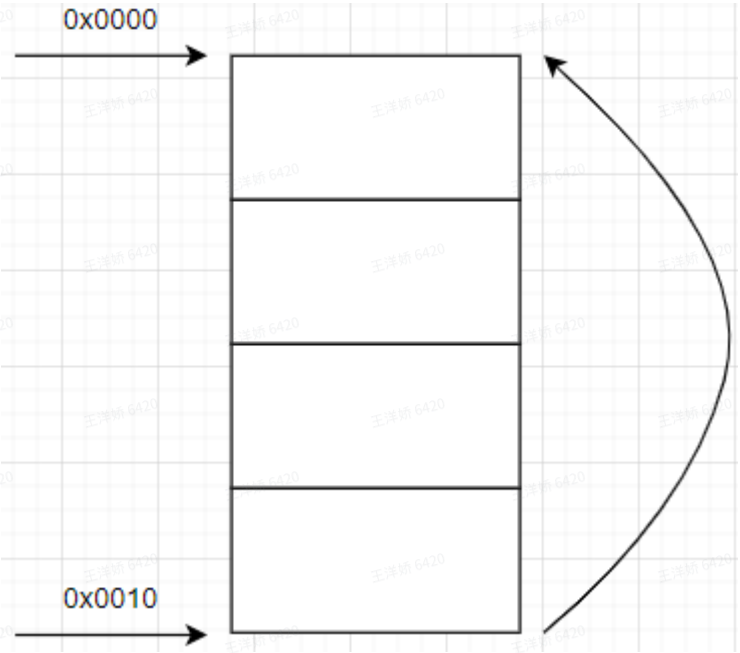
srcLastAddrAdjust

完成一次 Trigger Loop 后，源地址的地址偏移。以下图为例，此时 Access Loop 为4bytes，Transfer Loop 为16bytes，Trigger Loop 也为16bytes，即一次触发事件，DMA共搬运16bytes数据，源地址已至 0x0010

a. 若用户希望下一次触发，重新搬运 0x0000 地址的数据，则可将 srcLastAddrAdjust 设置为 -16

b. 若用户希望下一次触发，继续往下搬运，则可将 srcLastAddrAdjust 设置为 0

c. 若用户希望下一次触发，搬运 0x0020 地址的数据，则可将 srcLastAddrAdjust 设置为 16



destLastAddrAdjust

完成一次 Trigger Loop 后，目的地址的地址偏移。与 srcLastAddrAdjust 类似，不再赘述

srcModulo

源地址模值，可选值如下所示

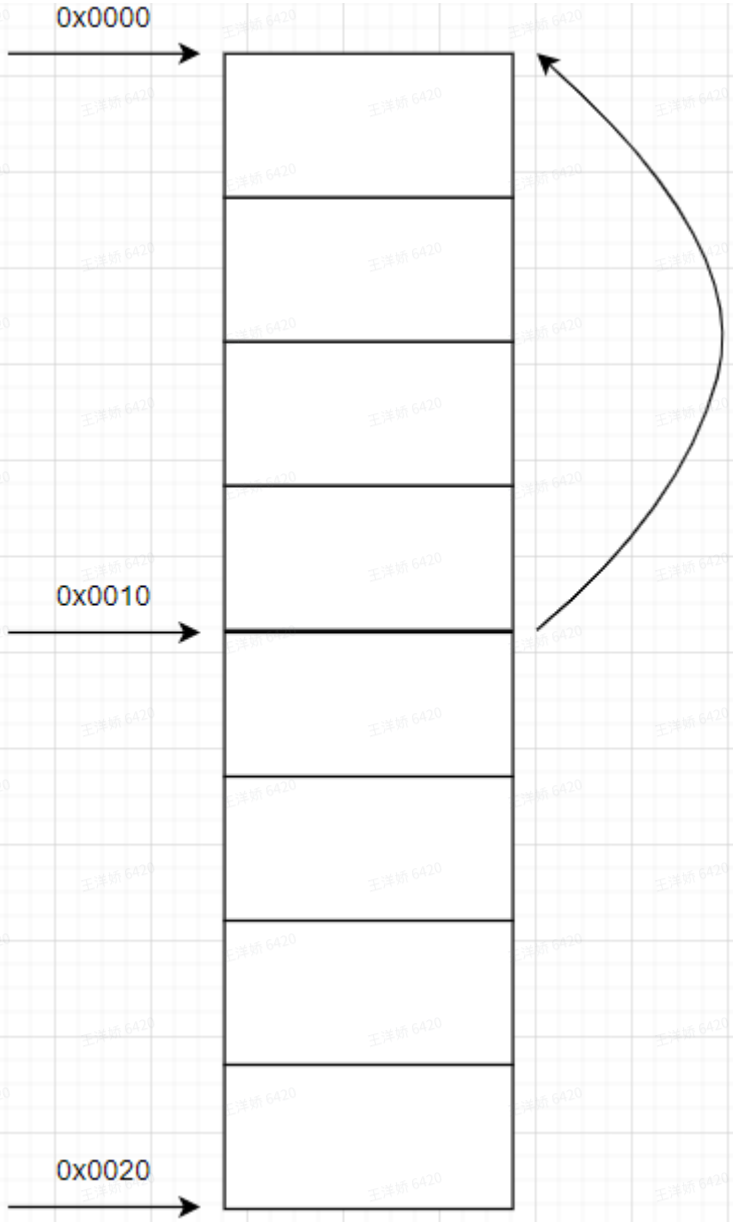
```
/*! @brief DMA modulo configuration
 * DMA source/destination address modulo value when address changes.
 * Implements : dma_modulo_t_Class
 */
typedef enum
{
    DMA_MODULO_OFF = 0U,
    DMA_MODULO_2B,
    DMA_MODULO_4B,
    DMA_MODULO_8B,
    DMA_MODULO_16B,
    DMA_MODULO_32B,
    DMA_MODULO_64B,
    DMA_MODULO_128B,
    DMA_MODULO_256B,
    DMA_MODULO_512B,
    DMA_MODULO_1KB,
    DMA_MODULO_2KB,
    DMA_MODULO_4KB,
    DMA_MODULO_8KB,
    DMA_MODULO_16KB,
    DMA_MODULO_32KB,
    DMA_MODULO_64KB,
```

```
24        DMA_MODULO_128KB,
25        DMA_MODULO_256KB,
26        DMA_MODULO_512KB,
27        DMA_MODULO_1MB,
28        DMA_MODULO_2MB,
29        DMA_MODULO_4MB,
30        DMA_MODULO_8MB,
31        DMA_MODULO_16MB,
32        DMA_MODULO_32MB,
33        DMA_MODULO_64MB,
34        DMA_MODULO_128MB,
35        DMA_MODULO_256MB,
36        DMA_MODULO_512MB,
37        DMA_MODULO_1GB,
38        DMA_MODULO_2GB
39    } dma_modulo_t;
```

以下图为例，此时 Access Loop 为4bytes，Transfer Loop 为32bytes

a. 若设置 srcModulo 为 *DMA_MODULO_OFF*，则一次触发事件后，DMA 会搬运 0x0000~0x0020 的所有数据

b. 若设置 srcModulo 为 *DMA_MODULO_16B*，则一次触发事件后，DMA 会搬运 0x0000~0x0010 的数据，随后重新搬运 0x0000~0x0010 的数据



更一般意义上来说，当 srcModulo 的设置为 *DMA_MODULO_16B* 时，源地址的 bit4-31 将不再发生变化，只有 bit0~3 可以变化。

destModulo

目的地址模值，与 srcModulo 类似，不再赘述

transferLoopByteCount

一次 Transfer Loop 所需搬运的字节数

ramReloadEnable

使能 Ram Reload，当完成 Trigger Loop 后，当前通道可以自动从内存中加载 DMA 的配置参数

ramReloadNextDescAddr

当完成 Trigger Loop 后，当前通道可以自动加载 DMA 的配置参数，内存地址即 ramReloadNextDescAddr

interruptEnable

使能 Trigger Loop 完成中断

loopTransferConfig

循环传输配置，该参数在下文详解

2.3 DMA 循环传输配置结构体介绍

```
/*!
 * @brief DMA loop transfer configuration.
 *
 * This structure configures the basic transfer/trigger loop attributes.
 * Implements : dma_loop_transfer_config_t_Class
 */
typedef struct
{
    uint32_t triggerLoopIterationCount;   /*!< Number of trigger loop iterations. */
    bool srcOffsetEnable;                 /*!< Selects whether the transfer loop offset is applied to the
                                               source address upon transfer loop completion. */
    bool dstOffsetEnable;                 /*!< Selects whether the transfer loop offset is applied to the
                                               destination address upon transfer loop completion. */
    int32_t triggerLoopOffset;            /*!< Sign-extended offset applied to the source or destination address
                                               to form the next-state value after the transfer loop completes. */
    bool transferLoopChnLinkEnable;       /*!< Enables channel-to-channel linking on transfer loop complete. */
    uint8_t transferLoopChnLinkNumber;    /*!< The number of the next channel to be started by DMA
                                               engine when transfer loop completes. */
    bool triggerLoopChnLinkEnable;        /*!< Enables channel-to-channel linking on trigger loop complete. */
    uint8_t triggerLoopChnLinkNumber;     /*!< The number of the next channel to be started by DMA
                                               engine when trigger loop completes. */
} dma_loop_transfer_config_t;
```

上图为 DMA 循环传输结构体

triggerLoopIterationCount

代表DMA 的一个通道总共需要响应多少次Trigger 事件，即 Trigger Loop

srcOffsetEnable

当完成一次 Transfer Loop 后，是否使能源地址偏移。注意，并非使能dma_transfer_config_t 结构体中的 srcOffset，srcOffset 为完成一次 Access Loop 后的目的地址偏移

dstOffsetEnable

当完成一次 Transfer Loop 后，是否使能目的地址偏移

triggerLoopOffset

完成一次 Transfer Loop 后，源地址与目的地址的偏移量，源地址与目的地址是否偏移受 srcOffsetEnable 与 dstOffsetEnable 控制。

transferLoopChnLinkEnabl

完成一次 Transfer Loop 后，使能通道链接功能

transferLoopChnLinkNumber

待链接的通道。Transfer Loop 通道链接功能：若当前通道完成一次 Transfer Loop 后，立即触发 transferLoopChnLinkNumber 该通道的传输，无需外部触发事件。

triggerLoopChnLinkEnable

完成一次 Trigger Loop 后，使能通道链接功能

triggerLoopChnLinkNumber

待链接的通道。TriggerLoop 通道链接功能：若当前通道完成一次 Trigger Loop 后，立即触发 transferLoopChnLinkNumber 该通道的传输，无需外部触发事件。

3. DMA 传输API介绍

3.1 DMA 基础配置

使用DMA 之前需要对DMA 进行初始化，这部分操作涉及DMA 整个模块的初始化和DMA 通道的初始化，可以通过如下函数进行初始化：

```
1  status_t DMA_DRV_Init(dma_state_t *dmaState,
2                        const dma_user_config_t *userConfig,
```

```
                    dma_chn_state_t *const chnStateArray[],
                    const dma_channel_config_t *const chnConfigArray[],
                    uint32_t chnCount)
```

- dmaState: 需要传入一个全局地址，初始化时候会保存DMA 通道的运行信息和回调信息，是一个dma_chn_state_t 指针的结构体数组。
- userConfig：定义了一些DMA 的全局配置，比如DMA 产生错误的时候是否继续进行传输
- chnStateArray：通道状态结构体数组，通道的State 也需要有全局地址，保存了通道的运行信息和回调，初始化过程中会将这个数组和dmaState 相关联。
- chnConfigArray：通道配置信息，包含对应的物理通道，通道传输完成回调配置和硬件触发通道配置。初始化过程中会将这个信息写入chnStateArray 进行保存。
- chnCount：是总共要初始化的channel 的数量

DMA_DRV_Init 函数需要和DMA_DRV_Deinit 配对使用，如果已经初始化了DMA，则需要用DMA_DRV_Deinit

函数先反初始化DMA 之后再重新初始化，否则可能出现意外的错误。

DMA 初始化完成之后就可以根据需要对DMA 的通道做数据传输的配置，并开启数据传输。对DMA 的数据传输分为简单的单块传输和灵活的多块传输，下面就对这两种数据传输配置进行简单介绍。

3.2 简单的单块传输

简单的单块的DMA 数据传输可以实现基础的数据传输操作，简单传输假定DMA 只需将数据从一个地方搬运到另一个地方，源地址和目的地址只有累加方式（Memory）和地址不变方式（Peripherals），DMA 可以有多次Trigger，每次搬运固定数量的数据。

数据传输的配置可以通过如下函数完成：

```
status_t DMA_DRV_ConfigSingleBlockTransfer(uint8_t virtualChannel,
                                           dma_transfer_type_t type,
                                           uint32_t srcAddr,
                                           uint32_t destAddr,
                                           dma_transfer_size_t transferSize,
                                           uint32_t dataBufferSize)
```

- virtualChannel：定义通过哪个通道进行传输，注意软件需要保证当前通道处于空闲状态，该函数并不会检

查通道的状态

- type：定义了数据的传输类型，分为 *DMA_TRANSFER_PERIPH2MEM*，*DMA_TRANSFER_MEM2PERIPH*，*DMA_TRANSFER_MEM2MEM*，*DMA_TRANSFER_PERIPH2PERIPH* 几种类型，函数通过该类型决定源地址和目的地址是否需要偏移
- srcAddr：源地址，DMA 从该地址开始（或者进访问该地址）读取数据
- destAddr：目的地址，DMA 将数据写入该地址的memroy 空间或者外设
- transferSize：定义了DMA 对源地址和目的地址数据操作的宽度，分为：*DMA_TRANSFER_SIZE_1B*，*DMA_TRANSFER_SIZE_2B*，*DMA_TRANSFER_SIZE_4B*，*DMA_TRANSFER_SIZE_16B*，*DMA_TRANSFER_SIZE_32B* 几种类型，注意MCU 可能只支持有限的几种类型，常用的是1B，2B 和4B，分别对应字节访问(8bit)，半字访问(16bit) 和字访问(32bit)。
- dataBufferSize：定义了单次传输的总字节数。

3.3 灵活的多块传输

针对一些复杂的传输，DMA 也提供了更为灵活的多块传输，可以实现更加复杂的数据传输，支持控制外部触发事件的数量，这样CPU 可以更少的介入外设数据传输从而降低CPU 的负载。多块传输通过如下函数配置：

```
status_t DMA_DRV_ConfigMultiBlockTransfer(uint8_t virtualChannel,
                                          dma_transfer_type_t type,
                                          uint32_t srcAddr,
                                          uint32_t destAddr,
                                          dma_transfer_size_t transferSize,
                                          uint32_t blockSize,
                                          uint32_t blockCount,
                                          bool disableReqOnCompletion)
```

多块传输的部分参数和单块传输相同，多块传输中引入了如下几个参数：

- blockSize：基本等同于dataBufferSize，代表DMA 每次Trigger 事件DMA 搬运数据的总字节数。
- blockCount：DMA 的Trigger Loop 的循环次数，也就是该DMA 通道总共接收多少个外部Trigger 事件。
- disableReqOnCompletion：设置是否在Trigger Loop 结束之后关闭通道的硬件触发，大部分情况下应该设置该参数为True，当该参数设置为False 时，如果DMA 已经完成所有的Trigger Loop，此时DMA 依然会响应外部的触发事件，DMA 会自动将Loop 次数设置为初始值（一般也就是Trigger Loop) 的数值，并开始新的传输。

3.4 更细致的传输

如果需要对DMA 的传输做更为细致的控制，可以通过如下的函数配置DMA 传输的额外属性：

```
status_t DMA_DRV_ConfigLoopTransfer(uint8_t virtualChannel,
                                    const dma_transfer_config_t
*transferConfig)
```

- transferConfig: DMA 传输配置结构体指针，即 2.2 章节中所提及的。

DMA_DRV_ConfigLoopTransfer 可以实现针对源地址和目的地址灵活的偏移设置，另外也可以设置通道和通道之

间的相互触发操作，另外还可以设置DMA 在完成当前Trigger Loop 之后从RAM 中重新load 新的配置信息。

4. DMA 应用示例

4.1 搬运 Flash 数据至 RAM

下面示例演示了 DMA 搬运 Flash 数据至 RAM

```
dma_state_t dmaController_State;
dma_chn_state_t dmaControllerChn0_State;

dma_chn_state_t * const dmaChnStateArray[] = {
    &dmaControllerChn0_State,
};

/* DMA user configuration */
dma_channel_config_t dmaControllerChn0_Config = {
    .virtChnConfig = 0,
    .source = DMA_REQ_Disabled,
    .callback = NULL,
    .callbackParam = NULL,
};

const dma_channel_config_t * const dmaChnConfigArray[] = {
    &dmaControllerChn0_Config,
};

const dma_user_config_t dmaController_InitConfig = {
    .haltOnError = false
};

#define MASS_DATA_BLOCK (64)
```

```c
volatile uint8_t done = 0;
uint8_t retData[MASS_DATA_BLOCK];
/* DMA callback */
void CompleteDMA(void *parameter, dma_chn_status_t status)
{
    done = 1;
    (void) parameter;
    if(DMA_CHN_ERROR == status)
    {
        PRINTF("Error happened!\n");
        while (1);
    }
}

void Mem2Mem(void)
{
    /* DMA Init */
    DMA_DRV_Init(&dmaController_State,
            &dmaController_InitConfig,
            dmaChnStateArray,
            dmaChnConfigArray,
            DMA_CONFIGURED_CHANNELS_COUNT);

    /* DMA carries data from 0x00000000 to retData */
    DMA_DRV_ConfigSingleBlockTransfer(0, DMA_TRANSFER_MEM2MEM, 0, (uint32_t)retData, DMA_TRANSFER_SIZE_4B,
                                      MASS_DATA_BLOCK);
    /* Install DMA callback */
    DMA_DRV_InstallCallback(0, CompleteDMA, NULL);
    /* Start DMA channel */
    DMA_DRV_StartChannel(0);
    /* Software trigger for DMA to start carrying data  */
    DMA_DRV_TriggerSwRequest(0);
    /* Wait DMA done */
    while(!done);
    PRINTF("DMA carries data from 0x00000000 to retData done\n");
}

```

上述示例演示了如何将 Flash 中的数据通过 DMA 搬运至 RAM。DMA 会搬运 0x0000_0000 的数据至 retData 数组中，共搬运 64bytes 数据。

4.2 SPI 用 DMA 方式发送数据

下面示例会演示如何通过DMA 方式实现 SPI 的数据发送

注意：只截取关键代码，以及只关心 DMA 的配置

```
dma_channel_config_t dmaControllerChn0_Config = {
    .virtChnConfig = 0,
    .source = DMA_REQ_SPI4_TX,
    .callback = NULL,
    .callbackParam = NULL,
};

...

#define SPI_TX_LEN  (100)
uint8_t spiTxBuf[SPI_TX_LEN];
void Mem2Periph(void)
{
    ...
    /* Config DMA for SPI transfer */
    DMA_DRV_ConfigMultiBlockTransfer(0, DMA_TRANSFER_MEM2PERIPH,
(uint32_t)spiTxBuf, (uint32_t)&(SPI4->DATA),
                                     DMA_TRANSFER_SIZE_1B, 1, SPI_TX_LEN,
true);
    /* Start DMA channel */
    DMA_DRV_StartChannel(0);
    ...
}
```

上述示例为 SPI 通过 DMA 方式发送数据的部分代码。主要内容如下：

1. 选择 DMA 通道的触发源，此例中为 DMA_REQ_SPI4_TX
2. 选择 DMA_DRV_ConfigMultiBlockTransfer 实现 SPI 发送数据。
3. 当 SPI 的发送 FIFO 为空时，产生DMA请求，DMA 会搬运 1byte 数据从 spiTxBuf 至 SPI 的发送 FIFO
4. 当前配置一次 Trigger 只搬运 1字节，要搬完 100 字节数据，需产生 100 个 Trigger

4.3 SPI 用 DMA 方式接收数据

下面示例会演示如何通过DMA 方式实现 SPI 的数据接收

注意：只截取关键代码，以及只关心 DMA 的配置

```
dma_channel_config_t dmaControllerChn0_Config = {
    .virtChnConfig = 1,
    .source = DMA_REQ_SPI4_RX,
```

```
    .callback = NULL,
    .callbackParam = NULL,
};

...

#define SPI_RX_LEN  (100)
uint8_t spiRxBuf[SPI_RX_LEN];
void Periph2Mem(void)
{
    ...
    /* Config DMA for SPI transfer */
    DMA_DRV_ConfigMultiBlockTransfer(1, DMA_TRANSFER_PERIPH2MEM, (uint32_t)&(SPI4->DATA), (uint32_t)spiRxBuf,
                                     DMA_TRANSFER_SIZE_1B, 1, SPI_RX_LEN, true);
    /* Start DMA channel */
    DMA_DRV_StartChannel(1);
    ...
}
```

上述示例为 SPI 通过 DMA 方式接收数据的部分代码。主要内容如下：

1. 选择 DMA 通道的触发源，此例中为 DMA_REQ_SPI4_RX
2. 选择 DMA_DRV_ConfigMultiBlockTransfer 实现 SPI 接收数据。
3. 当 SPI 的接收 FIFO 非空时，产生DMA请求，DMA 会搬运 1byte 数据从 SPI 的接收 FIFO 至 spiRxBuf
4. 当前配置一次 Trigger 只搬运 1字节，要搬完 100 字节数据，需产生 100 个 Trigger

4.4 实时刷新 ADC 近100次的转换结果

有一个 200 words 长度的数组，实时刷新 ADC 两个通道的近100次的转换结果。

```
#define ADC_LOOP_CNT                (100)
#define BUFFER_LENGTH               (200)
volatile uint32_t g_adc_result[BUFFER_LENGTH];          //in order to show AD channel ID

/* DMA loop transfer configuration for ADC */
dma_loop_transfer_config_t adcLoopTransferConfig = {
    .triggerLoopIterationCount = ADC_LOOP_CNT,          //how many transfer times for tansfer all datas.
    .dstOffsetEnable = false,
    .triggerLoopOffset = 0
```

```c
};

/* DMA transfer configuration for ADC */
const dma_transfer_config_t adcTransferConfig = {
    .srcAddr = (uint32_t)&ADC0->FIFO,                          // get ADC result
    .destAddr = (uint32_t)&g_adc_result[0U],                   // store ADC result
    .destOffset = 0x04U,                                       // 4bytes for ADC result
    .srcOffset = 0x00U,
    .srcTransferSize = DMA_TRANSFER_SIZE_4B,                   // DMA transfer 2Bytes one time.
    .destTransferSize = DMA_TRANSFER_SIZE_4B,
    .srcModulo = DMA_MODULO_OFF,
    .destModulo = DMA_MODULO_OFF,
    .transferLoopByteCount =  8,                               // Byte size per transfer.
    .loopTransferConfig = &adcLoopTransferConfig,
    .destLastAddrAdjust =  -(ADC_LOOP_CNT * 4),                // Refresh data buffer
    .interruptEnable = true                                    // Enable the interrupt request when the trigger loop count completed.
};

void AdcRefreshData(void)
{
    ...

    /* Config DMA for ADC */
    DMA_DRV_ConfigLoopTransfer(0, &adcTransferConfig);
    /* Start new trigger loop when trigger loop done */
    DMA_DRV_DisableRequestsOnTransferComplete(0, false);
    /* DMA channel */
    DMA_DRV_StartChannel(0);

    ...
}
```

上述示例为实时刷新 ADC 两个通道的近100次的转换结果。主要内容如下：

1. ADC 一个序列中有两个通道
2. 当 ADC 转换完成一个序列后，产生 DMA 请求，DMA 将 2words(8bytes) 数据搬运至 g_adc_result
3. 搬满 100 次记录需要共计产生 100 次 Trigger，每个 Trigger 搬运 8个字节数据
4. 使用 DMA_DRV_ConfigLoopTransfer 做 DMA 配置

5. 使用 DMA_DRV_DisableRequestsOnTransferComplete，可以让 DMA 在 Trigger Loop 完成之后，重新加载 Trigger Loop
6. .destLastAddrAdjust = -(ADC_LOOP_CNT * 4) 这个参数使得，当 Trigger Loop 完成后，目的地址刷回最开始的地址，可以刷新 ADC 两个通道的近100次的转换结果