

图：DMA Loop 测试源码结构

**1.重要的结构体**

　　1.1.中断设备：static **XScuGic** Intc; //GIC

　　在main.c文件中，sataic用来修饰全局变量，形成静态全局变量，static修饰的函数/全局变量属于内链接，Intc可以在当前main.c文件内部范围内进行链接。

[复制代码](javascript:void(0);)

/\*\*

\* The XScuGic driver instance data. The user is required to allocate a

\* variable of this type for every intc device in the system. A pointer

\* to a variable of this type is then passed to the driver API functions.

\*/

typedef struct

{

XScuGic\_Config \*Config; /\*\*< Configuration table entry \*/

u32 IsReady; /\*\*< Device is initialized and ready \*/

u32 UnhandledInterrupts; /\*\*< Intc Statistics \*/

} XScuGic;

[复制代码](javascript:void(0);)

　　XScuGic结构体中包含了XScuGic\_Config 结构体类型指针Config。XScuGic\_Config 结构体如下：

[复制代码](javascript:void(0);)

typedef struct

{

u16 DeviceId; /\*\*< Unique ID of device \*/

u32 CpuBaseAddress; /\*\*< CPU Interface Register base address \*/

u32 DistBaseAddress; /\*\*< Distributor Register base address \*/

XScuGic\_VectorTableEntry HandlerTable[XSCUGIC\_MAX\_NUM\_INTR\_INPUTS];/\*\*<

Vector table of interrupt handlers \*/

} XScuGic\_Config;

[复制代码](javascript:void(0);)

　　1.2.DMA设备：static  XAxiDma AxiDma;

[复制代码](javascript:void(0);)

/\*\*

\* The XAxiDma driver instance data. An instance must be allocated for each DMA

\* engine in use.

\*/

typedef struct XAxiDma {

UINTPTR RegBase; /\* Virtual base address of DMA engine \*/

int HasMm2S; /\* Has transmit channel \*/

int HasS2Mm; /\* Has receive channel \*/

int Initialized; /\* Driver has been initialized \*/

int HasSg;

XAxiDma\_BdRing TxBdRing; /\* BD container management for TX channel \*/

XAxiDma\_BdRing RxBdRing[16]; /\* BD container management for RX channel \*/

int TxNumChannels;

int RxNumChannels;

int MicroDmaMode;

int AddrWidth; /\*\*< Address Width \*/

} XAxiDma;

[复制代码](javascript:void(0);)

　　1.3.中断向量表

typedef struct {

Xil\_ExceptionHandler Handler;

void \*Data;

} XExc\_VectorTableEntry;

**2.代码结构**

main()

|---- init\_intr\_sys();

　　|---- DMA\_Intr\_Init();　　// 初始化DMA

　　　　|---- XAxiDma\_LookupConfig();　　// 查找DMA设备

　　　　|---- XAxiDma\_CfgInitialize();　　　　// 初始化DMA设备

　　|---- Init\_Intr\_System();　　//初始化中断控制器

　　　　|---- XScuGic\_LookupConfig();　　// 查找中断控制器设备；带的参数为设备ID，查看中断向量是否存在

　　　　|---- XScuGic\_CfgInitialize();　　// 初始化中断控制器设备

　　|---- Setup\_Intr\_Exception();

　　　　|---- Xil\_ExceptionInit();　　// 使能硬件中断

　　　　|---- Xil\_ExceptionRegisterHandler();

　　　　|---- Xil\_ExceptionEnable();

　　|---- DMA\_Setup\_Intr\_System();　　// 设置DMA中断

　　　　|---- XScuGic\_SetPriorityTriggerType();

　　　　|---- XScuGic\_Connect();　　// 连接中断源

　　　　|---- XScuGic\_Enable();

　　|---- DMA\_Intr\_Enable();

　　　　|---- XAxiDma\_IntrDisable();

　　　　|---- XAxiDma\_IntrEnable();

|----axi\_dma\_test();

2.1.比较重要的函数

2.1.1.中断注册函数 Xil\_ExceptionRegisterHandler：

[复制代码](javascript:void(0);)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* @brief Register a handler for a specific exception. This handler is being

\* called when the processor encounters the specified exception.

\*

\* @param exception\_id contains the ID of the exception source and should

\* be in the range of 0 to XIL\_EXCEPTION\_ID\_LAST.

\* See xil\_exception.h for further information.

\* @param Handler to the Handler for that exception.

\* @param Data is a reference to Data that will be passed to the

\* Handler when it gets called.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Xil\_ExceptionRegisterHandler(u32 Exception\_id,

Xil\_ExceptionHandler Handler,

void \*Data)

{

XExc\_VectorTable[Exception\_id].Handler = Handler;

XExc\_VectorTable[Exception\_id].Data = Data;

}

[复制代码](javascript:void(0);)

　　从上面可以看到Xil\_ExceptionRegisterHandler()这个函数是把中断的句柄(第二个行参“Handler”)和中断的参数(第三个行参“Data”)放到了两个结构体XExc\_VectorTableEntry类型的数组XExc\_VectorTable当中 ，XExc\_VectorTableEntry结构体类型如下：

[复制代码](javascript:void(0);)

XExc\_VectorTableEntry XExc\_VectorTable[XIL\_EXCEPTION\_ID\_LAST + 1] =

{

{Xil\_ExceptionNullHandler, NULL},

{Xil\_UndefinedExceptionHandler, NULL},

{Xil\_ExceptionNullHandler, NULL},

{Xil\_PrefetchAbortHandler, NULL},

{Xil\_DataAbortHandler, NULL},

{Xil\_ExceptionNullHandler, NULL},

{Xil\_ExceptionNullHandler, NULL},

};

[复制代码](javascript:void(0);)

　　Xil\_ExceptionRegisterHandler()函数的第二传参XScuGic\_InterruptHandler，是一个函数指针，强制转化成了Xil\_ExceptionHandler类型，XScuGic\_InterruptHandler()函数如下。

[复制代码](javascript:void(0);)

void XScuGic\_InterruptHandler(XScuGic \*InstancePtr)

{

u32 InterruptID;

u32 IntIDFull;

XScuGic\_VectorTableEntry \*TablePtr;

/\* Assert that the pointer to the instance is valid

\*/

Xil\_AssertVoid(InstancePtr != NULL);

/\*

\* Read the int\_ack register to identify the highest priority interrupt ID

\* and make sure it is valid. Reading Int\_Ack will clear the interrupt in the GIC.

\* 读取 int\_ack 寄存器以识别最高优先级的中断 ID， 并确保其有效。读取 Int\_Ack 将清除 GIC 中的中断。  
　　　　　\*　然后看看读出来的中断 ID 是否大于最大的中断值。

\*/

IntIDFull = XScuGic\_CPUReadReg(InstancePtr, XSCUGIC\_INT\_ACK\_OFFSET);

InterruptID = IntIDFull & XSCUGIC\_ACK\_INTID\_MASK;

if(XSCUGIC\_MAX\_NUM\_INTR\_INPUTS < InterruptID){

goto IntrExit;

}

/\*

\* Execute the ISR. Jump into the Interrupt service routine based on the

\* IRQSource. A software trigger is cleared by the ACK.

\*/

TablePtr = &(InstancePtr->Config->HandlerTable[InterruptID]);

if(TablePtr != NULL) {

TablePtr->Handler(TablePtr->CallBackRef);

}

IntrExit:

/\*

\* Write to the EOI register, we are all done here.

\* Let this function return, the boot code will restore the stack.

\*/

XScuGic\_CPUWriteReg(InstancePtr, XSCUGIC\_EOI\_OFFSET, IntIDFull);

}

[复制代码](javascript:void(0);)

　　通过程序开头 xilinx 给出的这个XScuGic\_InterruptHandler()程序的注释可以知道： 这个函数是基本的中断驱动函数。 它必须 连接到中断源， 以便在中断控制器的中断激活时被调用。 它将解决哪些中断是活动的和启用的， 并调用适当的中断处理程序。 它使用中断类型信息来确定何时确认中断。 首先处理最高优先级的中断。 此函数假定中断向量表已预先初始化。 它不会在调用中断处理程序之前验证表中的条目是否有效。 **当中断发生时，调用的就是上面的代码中的语句：TablePtr->Handler(TablePtr->CallBackRef)。**那么这个Handler和CallBackRef到底是什么呢？也就是Handler和CallBackRef到底是和哪段要被执行的代码绑定在一起呢？  
2.1.2.中断连接函数

　　我们在DMA\_Setup\_Intr\_System()函数中调用了中断连接函数XScuGic\_Connect();

[复制代码](javascript:void(0);)

int DMA\_Setup\_Intr\_System(XScuGic \* IntcInstancePtr,XAxiDma \* AxiDmaPtr, u16 TxIntrId, u16 RxIntrId)

{

<...>

/\*

\* Connect the device driver handler that will be called when an

\* interrupt for the device occurs, the handler defined above performs

\* the specific interrupt processing for the device.

\*/

Status = XScuGic\_Connect(IntcInstancePtr, TxIntrId,

(Xil\_InterruptHandler)DMA\_TxIntrHandler,

AxiDmaPtr);

if (Status != XST\_SUCCESS) {

return Status;

}

Status = XScuGic\_Connect(IntcInstancePtr, RxIntrId,

(Xil\_InterruptHandler)DMA\_RxIntrHandler,

AxiDmaPtr);

if (Status != XST\_SUCCESS) {

return Status;

}

<...>

}

[复制代码](javascript:void(0);)

　　可以看到XScuGic\_Connect()函数的第三个传参是一个Xil\_InterruptHandler类型的函数指针DMA\_TxIntrHandler。XScuGic\_Connect()内容如下。

[复制代码](javascript:void(0);)

s32 XScuGic\_Connect(XScuGic \*InstancePtr, u32 Int\_Id,

Xil\_InterruptHandler Handler, void \*CallBackRef)

{

/\*

\* Assert the arguments

\*/

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(Int\_Id < XSCUGIC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertNonvoid(Handler != NULL);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* The Int\_Id is used as an index into the table to select the proper

\* handler

\*/

InstancePtr->Config->HandlerTable[Int\_Id].Handler = Handler;

InstancePtr->Config->HandlerTable[Int\_Id].CallBackRef = CallBackRef;

return XST\_SUCCESS;

}

[复制代码](javascript:void(0);)

　　可以看到XScuGic\_Connect()函数将传进来的第三个参数，Xil\_InterruptHandler类型的“Handler”，绑定到InstancePtr->Config->HandlerTable[Int\_Id].Handler中，第四个参数同理绑定。这里的InstancePtr是函数XScuGic\_Connect()传进来的XScuGic结构体类型的指针变量，前面讲过，XScuGic结构体中还包含XScuGic\_Config结构体类型的指针Config，进一步来说，XScuGic\_Connect()函数将传进来的第三个参数Handler就是绑定到XScuGic\_Config结构体类型的指针Config中的HandlerTable变量。这个HandlerTable变量是一个XScuGic\_VectorTableEntry类型的结构体变量。**至此，中断的Handler就绑定到main.c文件开头定义的设备static XScuGic Intc当中，同时设备 XScuGic Intc也因为函数Setup\_Intr\_Exception()跟硬件的异常向量表绑定到一起了。**前文提到“Handler和CallBackRef到底是和哪段要被执行的代码绑定在一起呢？”，那么答案就在这里了，要执行的代码就在这里被绑定到一起来。

　　所以接下来看看这个形参Handler（对应的是调用XScuGic\_Connect()函数传进来的实参DMA\_TxIntrHandler）指向了什么东西？

　　函数指针DMA\_TxIntrHandler指向的内容如下。下面代码是DMA Tx的，Rx的也差不多。

[复制代码](javascript:void(0);)

static void DMA\_TxIntrHandler(void \*Callback)

{

u32 IrqStatus;

int TimeOut;

XAxiDma \*AxiDmaInst = (XAxiDma \*)Callback;

/\* Read pending interrupts \*/

IrqStatus = XAxiDma\_IntrGetIrq(AxiDmaInst, XAXIDMA\_DMA\_TO\_DEVICE);

/\* Acknowledge pending interrupts \*/

XAxiDma\_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA\_DMA\_TO\_DEVICE);

/\*

\* If no interrupt is asserted, we do not do anything

\*/

if (!(IrqStatus & XAXIDMA\_IRQ\_ALL\_MASK)) {

return;

}

/\*

\* If error interrupt is asserted, raise error flag, reset the

\* hardware to recover from the error, and return with no further

\* processing.

\*/

if ((IrqStatus & XAXIDMA\_IRQ\_ERROR\_MASK)) {

Error = 1;

/\*

\* Reset should never fail for transmit channel

\*/

XAxiDma\_Reset(AxiDmaInst);

TimeOut = RESET\_TIMEOUT\_COUNTER;

while (TimeOut) {

if (XAxiDma\_ResetIsDone(AxiDmaInst)) {

break;

}

TimeOut -= 1;

}

return;

}

/\*

\* If Completion interrupt is asserted, then set the TxDone flag

\*/

if ((IrqStatus & XAXIDMA\_IRQ\_IOC\_MASK)) {

TxDone = 1;

}

}

[复制代码](javascript:void(0);)

 　　代码中，在dma\_intr.h中声明了三个全局变量：TxDone，RxDone和Error。

dam\_intr.h ：

extern volatile int TxDone;

extern volatile int RxDone;

extern volatile int Error;

　　在dma\_intr.c文件的开头，定义了这三个全局变量。

#include "dma\_intr.h"

volatile int TxDone;

volatile int RxDone;

volatile int Error;

可以看到用关键字volatile修饰，在此回顾一下朱有鹏老师在讲解C语言的时候，总结的volatile的用法：

(1)volatile的字面意思：可变的、易变的。C语言中volatile用来修饰一个变量，表示这个变量可以被编译器之外的东西改变。编译器之内的意思是变量的值的改变是代码的作用，编译器之外的改变就是这个改变不是代码造成的，或者不是当前代码造成的，编译器在编译当前代码时无法预知。譬如在中断处理程序isr中更改了这个变量的值，譬如多线程中在别的线程更改了这个变量的值，譬如硬件自动更改了这个变量的值（一般这个变量是一个寄存器的值）  
(2)以上说的三种情况（中断isr中引用的变量，多线程中共用的变量，硬件会更改的变量）都是编译器在编译时无法预知的更改，此时应用使用volatile告诉编译器这个变量属于这种（可变的、易变的）情况。编译器在遇到volatile修饰的变量时就不会对改变量的访问进行优化，就不会出现错误。  
(3)编译器的优化在一般情况下非常好，可以帮助提升程序效率。但是在特殊情况（volatile）下，变量会被编译器想象之外的力量所改变，此时如果编译器没有意识到而去优化则就会造成优化错误，优化错误就会带来执行时错误。而且这种错误很难被发现。  
(4)volatile是程序员意识到需要volatile然后在定义变量时加上volatile，如果你遇到了应该加volatile的情况而没有加程序可能会被错误的优化。如果在不应该加volatile而加了的情况程序不会出错只是会降低效率。所以我们对于volatile的态度应该是：正确区分，该加的时候加不该加的时候不加，如果不能确定该不该加为了保险起见就加上。