第七章 C 语言编程与中断

在第三章，我们介绍了 Cortex-M0 相关的异常与中断的处理过程。本章以键盘模块实验为基础，讲解 CPU 的中断处理以及使用 C 语言高效地编程。更多具体的细节可以参看第三章，现在我们简述中断的处理过程。

根据 ARMv6-M 架构参考手册以及 Cortex-M0 用户手册，CPU 中断处理过程如下：

- CPU 接收到中断信号（IRQ、NMI、Systick 等等）；
- 将 R0,R1,R2,R3,R12,LR,PC,xPSR 寄存器入栈，如图 7-1 所示；
- 根据中断信号查找中断向量表（对应汇编启动代码中的__Vector 段），跳转至中断处理函数，如图 7-2 所示；
- 中断处理函数执行完成后，利用链接寄存器返回，寄存器出栈，PC 跳转。

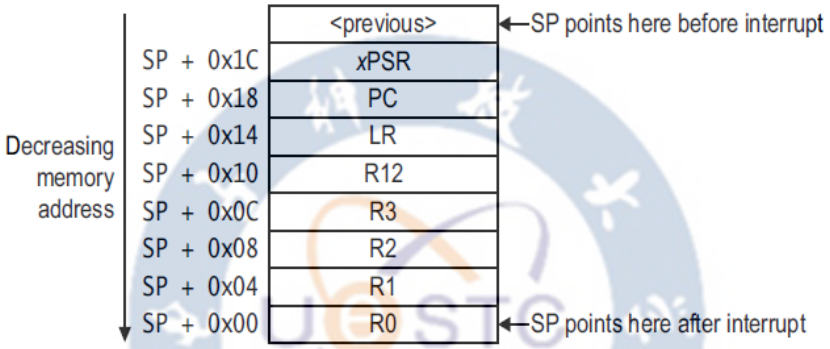

图 7- 1 寄存器入栈

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[e]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[e]	0x00000038	Asynchronous
15	-1	SysTick[c]	Configurable[e]	0x0000003C	Asynchronous
15	-	Reserved	-	-	-
16 and above[d]	0 and above	IRQ	Configurable[e]	0x00000040 and above[f]	Asynchronous

图 7- 2 异常中断向量表

本章最终实现的 SoC 如图 7-3 所示。利用开发板上面的矩阵键盘最下面的 4 个按键，通过上升沿触发 Cortex-M0 的 IRQ 中断。然后处理器在中断服务程序中，控制硬件流水灯的不同模式。

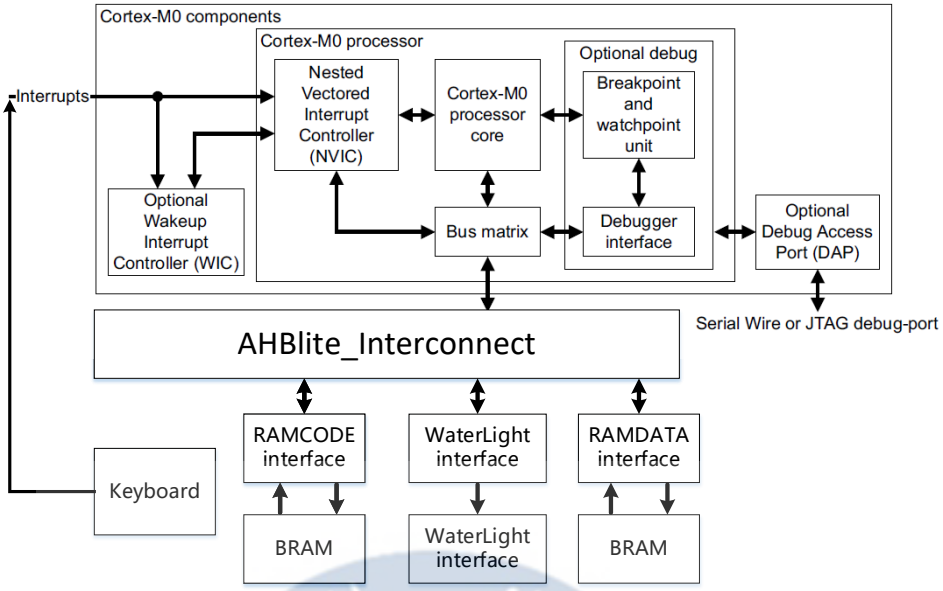

图 7- 3 本次实验的 SoC

7.1 硬件部分

7.1.1 矩阵键盘原理

在介绍矩阵键盘之前，我们先来介绍独立按键的检查原理。按键是一个简单的基础外设，只需要检测按键上的电平信号，就可以判断按键是否被按下。需要注意的是，如图 7-4 所示在按下按键的过程，会出现电平是不稳定的情况，持续时间大概为 1-10ms，这就有可能会触发多次按键按下的信号，因此需要对按键经行消抖处理。本次我们采取消抖方案为连续若干个周期检查到有效电平就认定按键按下。

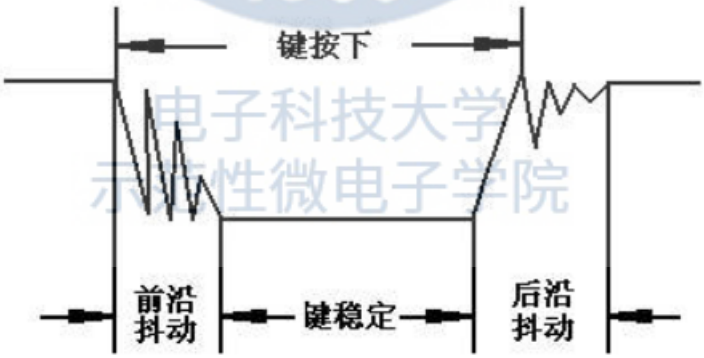

图 7- 4 按键按下的电平变化

本次实验所使用的开发板上有一个 4 x 4 的矩阵键盘，其原理图如图 7-5 所示。矩阵键盘的使用是为了减少键盘对芯片输入/输出端口的使用，如图所示控制 16 个按键只需要 8 个端口口，而如果每个按键占用一个端口就需要 16 个端口。矩阵键盘的驱动原理就比独立按键稍微复杂一些。矩阵键盘采用的是扫描方式进行检查，即依次让矩阵键盘的行线输出改变，然后检查列线的电平变化，这样就能定位被按下按键的坐标。

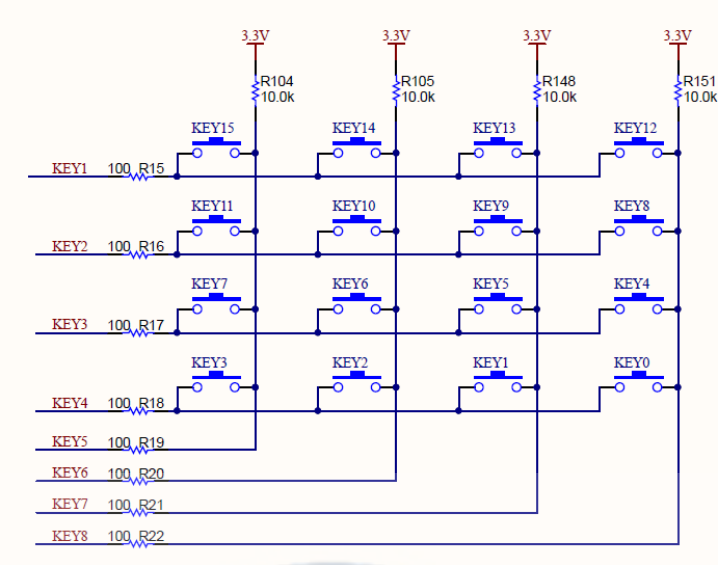

图 7- 5 矩阵键盘原理图

本次实验只使用了 KEY0-KEY3 因此不需要扫描，只需保持 KEY4 这根线为低电平即可。

7.1.2 硬件代码的修改

本章中，我们需要把 4 个按键分别分配到 Cortex-M0 的 IRQ0-IRQ3 四个中断上，按键的电平变化就会触发 Cortex-M0 对于的中断。

第一步，将按键模块输出的信号接到 Cortex-M0 系统的 IRQ 信号上，因此在 CortexM0_SoC.v 文件做如下修改。

```
/*Connect the IRQ with keyboard*/
assign IRQ = 32'b0;
/***************************/
```

改为：

```
/*Connect the IRQ with keyboard*/
assign IRQ = {28'b0,key_interrupt};
/***************************/
```

接下来，我们在 WaterLight 实验的 vivado 约束文件的基础上添加按键端口的约束。

```
##led
set_property PACKAGE_PIN P9 [get_ports {LED[0]}]
set_property PACKAGE_PIN R8 [get_ports {LED[1]}]
set_property PACKAGE_PIN R7 [get_ports {LED[2]}]
set_property PACKAGE_PIN T5 [get_ports {LED[3]}]
set_property PACKAGE_PIN N6 [get_ports {LED[4]}]
set_property PACKAGE_PIN T4 [get_ports {LED[5]}]
set_property PACKAGE_PIN T3 [get_ports {LED[6]}]
set_property PACKAGE_PIN T2 [get_ports {LED[7]}]
set_property PACKAGE_PIN R1 [get_ports LEDclk]

set_property IOSTANDARD LVCMOS33 [get_ports LEDclk]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[7]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]

##clk
set_property PACKAGE_PIN D4 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

##RSTn
set_property PACKAGE_PIN T9 [get_ports RSTn]
set_property IOSTANDARD LVCMOS33 [get_ports RSTn]

##DEBUGGER
set_property PACKAGE_PIN H14 [get_ports SWDIO]
set_property IOSTANDARD LVCMOS33 [get_ports SWDIO]
set_property PACKAGE_PIN H12 [get_ports SWCLK]
set_property IOSTANDARD LVCMOS33 [get_ports SWCLK]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SWCLK]

##led
set_property PACKAGE_PIN P9 [get_ports {LED[0]}]
set_property PACKAGE_PIN R8 [get_ports {LED[1]}]
set_property PACKAGE_PIN R7 [get_ports {LED[2]}]
set_property PACKAGE_PIN T5 [get_ports {LED[3]}]
set_property PACKAGE_PIN N6 [get_ports {LED[4]}]
set_property PACKAGE_PIN T4 [get_ports {LED[5]}]
set_property PACKAGE_PIN T3 [get_ports {LED[6]}]
set_property PACKAGE_PIN T2 [get_ports {LED[7]}]
set_property PACKAGE_PIN R1 [get_ports LEDclk]

set_property IOSTANDARD LVCMOS33 [get_ports LEDclk]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
```

```
##keyboard
set_property PACKAGE_PIN K3 [get_ports row[0]]
set_property IOSTANDARD LVCMOS33 [get_ports row[0]]
set_property PACKAGE_PIN M6 [get_ports row[1]]
set_property IOSTANDARD LVCMOS33 [get_ports row[1]]
set_property PACKAGE_PIN P10 [get_ports row[2]]
set_property IOSTANDARD LVCMOS33 [get_ports row[2]]
set_property PACKAGE_PIN R10 [get_ports row[3]]
set_property IOSTANDARD LVCMOS33 [get_ports row[3]]

set_property PACKAGE_PIN T10 [get_ports col[0]]
set_property IOSTANDARD LVCMOS33 [get_ports col[0]]
set_property PACKAGE_PIN R11 [get_ports col[1]]
set_property IOSTANDARD LVCMOS33 [get_ports col[1]]
set_property PACKAGE_PIN T12 [get_ports col[2]]
set_property IOSTANDARD LVCMOS33 [get_ports col[2]]
set_property PACKAGE_PIN R12 [get_ports col[3]]
set_property IOSTANDARD LVCMOS33 [get_ports col[3]]
```

7.2 启动代码与 C 编程

我们需要根据 CMSIS 提供的启动代码重新完成自己的启动代码，具体代码见“/Task4/keil/startup_CMSDK_CM0.s”。

与之前的汇编代码不同的是，我们在复位处理函数内调用了__mian 函数，此函数的作用是将堆栈初始化后跳转至 C 语言中的 mian 函数，而最后一段__user_initial_stackheap 则是初始化堆栈过程的一部分。初始化堆栈的具体过程由编译器提供，无需人为添加。

在中断处理的地方可以看到，当按键中断发生后，CPU 会根据__Vector 中的中断地址跳转到按键中断处理函数，在这个函数里面，首先人为地将寄存器入栈，然后跳转至 C 语言中的 key 函数，执行完成后寄存器出栈并返回。

先修改__Vector 中断向量表如下：

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     0                         ; NMI Handler
                DCD     0                         ; Hard Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; SVCall Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
```

```
                DCD     0                       ; PendSV Handler
                DCD     0                       ; SysTick Handler
                DCD     KEY0_Handler            ; IRQ0 Handler
                DCD     KEY1_Handler            ; IRQ1 Handler
                DCD     KEY2_Handler            ; IRQ2 Handler
                DCD     KEY3_Handler            ; IRQ3 Handler
```

添加中断复位函数的入口，如下：

```
; add IRQ Handler function here


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

改为：

```
KEY0_Handler    PROC
                EXPORT KEY0_Handler           [WEAK]
                IMPORT KEY0
                 PUSH    {R0,R1,R2,LR}
                 BL      KEY0
                 POP     {R0,R1,R2,PC}
                 ENDP

KEY1_Handler    PROC
                EXPORT KEY1_Handler           [WEAK]
                IMPORT KEY1
                 PUSH    {R0,R1,R2,LR}
                 BL      KEY1
                 POP     {R0,R1,R2,PC}
                 ENDP

KEY2_Handler    PROC
                 EXPORT KEY2_Handler           [WEAK]
                 IMPORT KEY2
                 PUSH    {R0,R1,R2,LR}
                 BL      KEY2
                 POP     {R0,R1,R2,PC}
                 ENDP

KEY3_Handler    PROC
                 EXPORT KEY3_Handler           [WEAK]
                 IMPORT KEY3
                 PUSH    {R0,R1,R2,LR}
                 BL      KEY3
                 POP     {R0,R1,R2,PC}
                 ENDP
```

然后，我们需要定义外设的地址，以及自己实现的函数，参考 CMSIS 编写自己头文件。具体代码见“/Task4/keil/code_def.h”。

```
#include <stdint.h>
//INTERRUPT DEF
#define NVIC_CTRL_ADDR (*(volatile unsigned *)0xe000e100)
//WATERLIGHT DEF
typedef struct{
   volatile uint32_t WaterLight_MODE;
   volatile uint32_t WaterLight_SPEED;
}WaterLightType;
#define WaterLight_BASE 0x40000000
#define WaterLight ((WaterLightType *)WaterLight_BASE)

void SetWaterLightMode(int mode);
void SetWaterLightSpeed(int speed);
```

第一行<stdint.h>头文件提供了结构体以及结构体运算符"->"的支持，高效地利用结构体定义外设地址，能够有效地减少代码量，节约存储空间。

下面以 WaterLight 为例讲解结构体与基地址的使用。首先我们根据之前 WaterLight 硬件部分设计，WaterLight 在地址空间能有两个个寄存器，分别为 Waterlight_MODE、Waterlight_SPEED，它们的地址分别为 0x40000000、0x40000004。两个寄存器在内存空间中是连续的两个字（word），因此在结构体中定义两个寄存器时需要按照它们地址的顺序依次定义，并且类型为 32bit 的 uint32_t。之后再定义 WaterLight 的基地址为 0x40000000。这样一来，当我们使用结构体中第一个元素时，它的地址则为基地址+0；第二个地址为基地址+4；第三个地址为基地址+8 依次类推。完全符合我们在硬件时定义的地址。

然后，我们需要完成函数的实现，具体见"/Task4/keil/code_def.c"。

```
#include "code_def.h"
#include <string.h>

void SetWaterLightMode(int mode)
{
    WaterLight -> Waterlight_MODE = mode;
}

void SetWaterLightSpeed(int speed)
{
    WaterLight -> Waterlight_SPEED = speed;
}
```

在有了这些函数以后，我们根据 startup_CMSDK_CM0.s 启动文件中所写的按键中断服务函数的名称，在 keyboard.c 中补充完整对应的中断服务函数。按键中断服务函数中，每个按键都对应一种流水灯的模式。

```
#include <stdint.h>
#include "code_def.h"

void KEY0(void)
{
}
```

```
void KEY1(void)
{
}

void KEY2(void)
{
}

void KEY3(void)
{
}
```

改为：

```
#include <stdint.h>
#include "code_def.h"
void KEY0(void)
{
    SetWaterLightMode(0);
}

void KEY1(void)
{
    SetWaterLightMode(1);
}

void KEY2(void)
{
    SetWaterLightMode(2);
}

void KEY3(void)
{
    SetWaterLightMode(3);
}
```

最后，编写主文件,具体在“/Task4/keil/main.c”。需要注意的是，在中断开启之前我们需要使能所用到的中断。

```
#include "code_def.h"
#include <string.h>
#include <stdint.h>

#define WaterLight_SPEED_VALUE 0x00c9d2ff

int main()
{
```

```
        //interrupt initial
        NVIC_CTRL_ADDR = 0xf;

        //WATERLIGHT
        SetWaterLightSpeed(WaterLight_SPEED_VALUE);
        while(1){
        }
    }
```

7.3 调试与运行

打开 Keil 工程将编写好的文件添加至工程中，并在如下图所示的设置中取消勾选“Don’t Search Standard Libraries”，然后编译，如图 7-6。

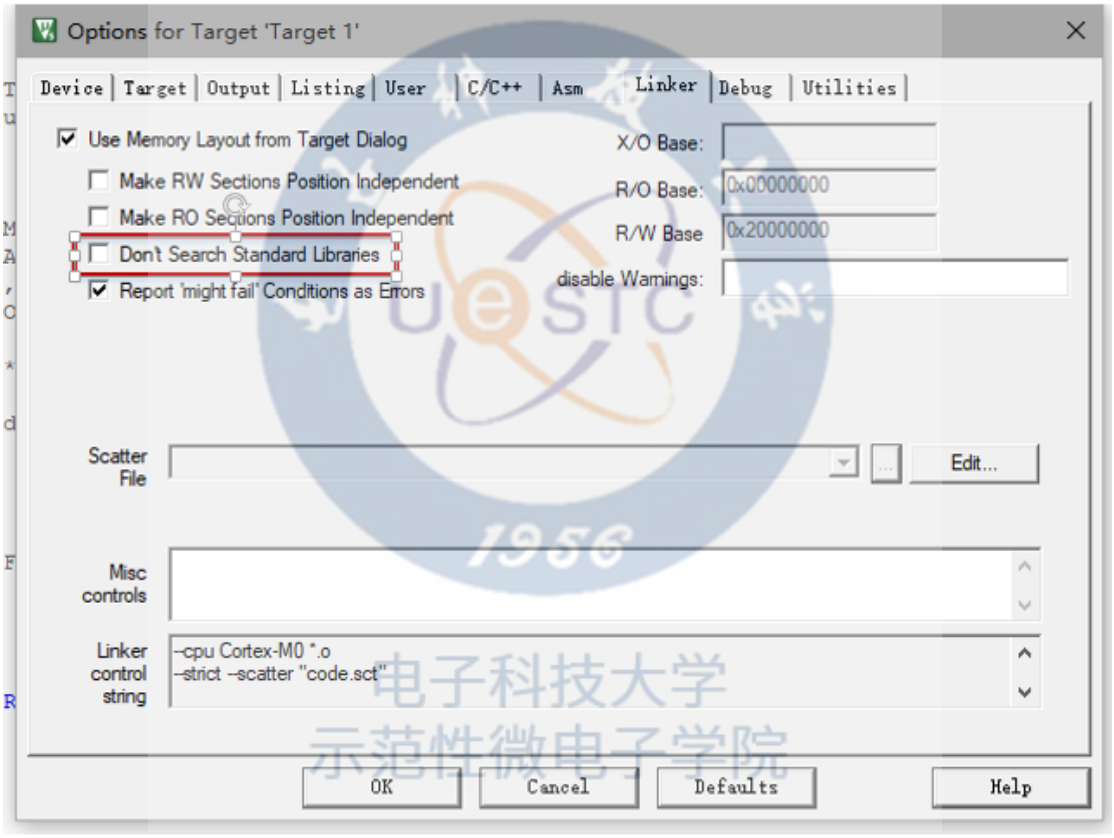

图 7- 6 取消勾选

在软件编译通过之后，我们使用 modelsim 进行仿真，我们在 testbench 文件中添加了一个按键信号用于触发处理器的中断，在触发中断后，我们能够观察处理器内部的变化。在 object 界面，我们选择添加 clk, col 按键输入端口，以及处理器内核的 IPSR 中断程序状态寄存器和 PC 寄存器 vis_ipsr_o。可以看到在按键输入保持一段时候有效后，IPSR 的值就会变为 16，根据第三章所述，IPSR 为记录异常标号的寄存器。此时 IPSR 为 16，查找中断向量表可知，此时处理器接收到了 IRQ0 中断。

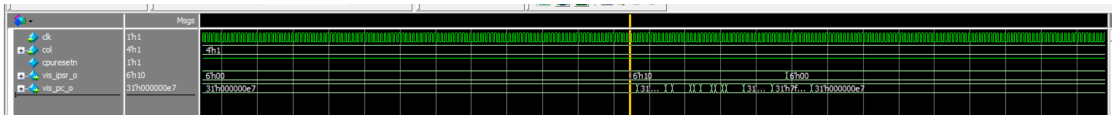

图 7- 7 modelsim 仿真

在 modelsim 仿真通过之后，我们将相关的文件添加到 vivado 工程中，最后将 vivado 生成的 bit 流文件下载到 FPGA 开发板上。我们就能够通过按键控制硬件流水灯的模式。

至此，我们就完成了本章的实验内容。

7.4 本章小结

本章中，我们用简单的例子实现了 Cortex-M0 的中断功能。通过实验过程，我们能够观察到处理器处理中断的过程，以及处理器内部的变化。并且在本章中，我们介绍了 C 语言高效编程。我们通过修改启动文件 startup_CMSDK_CM0.s，就能够实现使用 C 语言对处理器进行编程。同时我们也利用了 C 语言中结构体的特性，对我们设计的硬件进行很好的映射。