

本科实验报告

课程名称： **FPGA应用**

主讲教师： 夏书峰

学院、系： 微电子学院

专 业： 电子科学与技术类

班 级：

学 号：

姓 名：

联系电话：

电子邮箱：

2021～2022学年第3学期

实验成绩记录表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 内容 | 学时 | 满分 | 得分 |
| 1 | Vivado/ModelSim实践 | 4 | 15 |  |
| 2 | 时序逻辑设计1 — RTC | 4 | 20 |  |
| 3 | Xilinx嵌入式逻辑分析仪 | 4 | 15 |  |
| 4 | Xilinx嵌入式处理器实践 | 4 | 20 |  |
| 5 | 时序逻辑设计2 — FSM | 8 | 30 |  |
|  |  |  |  |  |
| 总计 | | 24 | 100 |  |

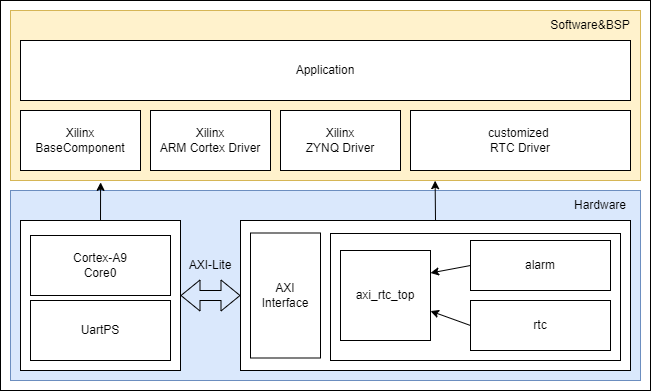
主讲教师签字：

实验四《Xilinx ZYNQ嵌入式处理器实践》实验报告

【实验内容】

硬件实现RTC：改造实验二中的RTC逻辑，封装成AXI4-Lite IP，把RTC计时寄存器的输出映射给AXI IP的内部寄存器，通过在PS里编程访问RTC的寄存器，获取当前走时时间，实现和软件方式同样的功能。自定义IP中预留一些内部寄存器，由PS写入要设置的时间，再通过某种方式（比如写某个寄存器的某 bit）操纵RTC的LOAD信号加载新时间，LOAD信号解除后，RTC就从当前时间开始走下去了。在RTC里添加硬件逻辑实现闹钟ALARM功能。

【本项目实现功能】



**硬件RTC实现**：使用Xilinx提供AXI-Lite总线接口封装实验二中的IP，并在其基础上挂载闹钟ALARM逻辑，对外提供只读的时钟输出寄存器接口和可读写的RTC配置寄存器接口，让PS端Cortex-A9硬核能够访问硬件RTC模块

**AXI-GPIO硬件实现**：使用Xilinx提供的AXI-GPIO为PS端转接控制板载LED和按钮的输出/输入接口

**RTC硬件抽象层驱动**：基于硬件抽象思想为RTC设计了一套便于移植和使用的硬件抽象层驱动，上层应用可以忽略底层寄存器操作对RTC进行控制

**串口输出当前时间功能**：串口以题目指定2022-08-17 THU 15:10:30的格式输出当前时间

**题目要求RTC基本功能**：RTC当前时间可由计算机发送命令修改，能通过命令设定 1 组闹钟时间，板载LED可以配合 RTC 闹钟点亮

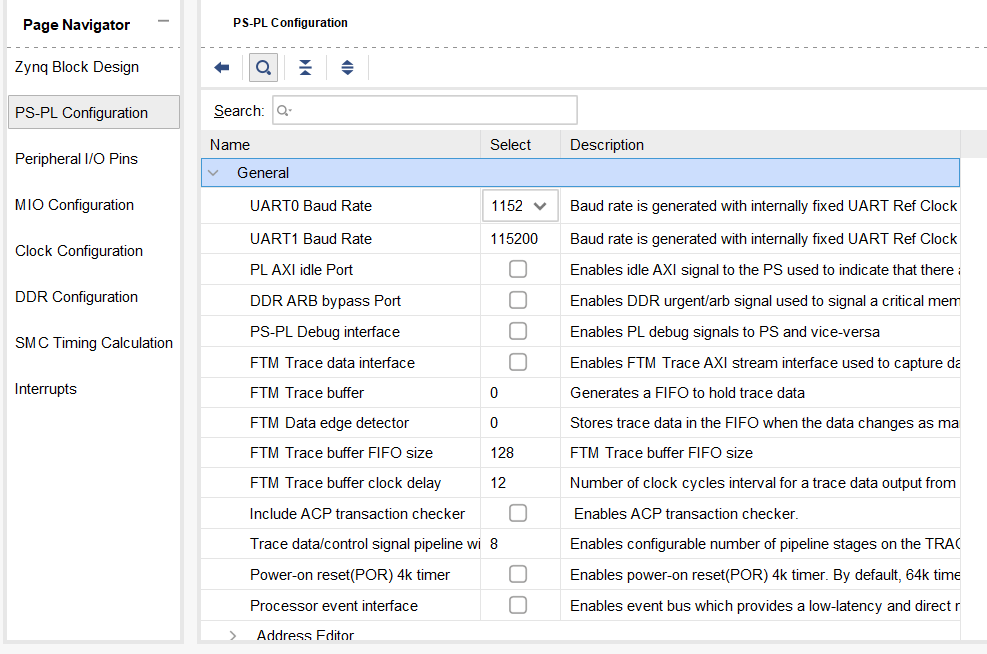
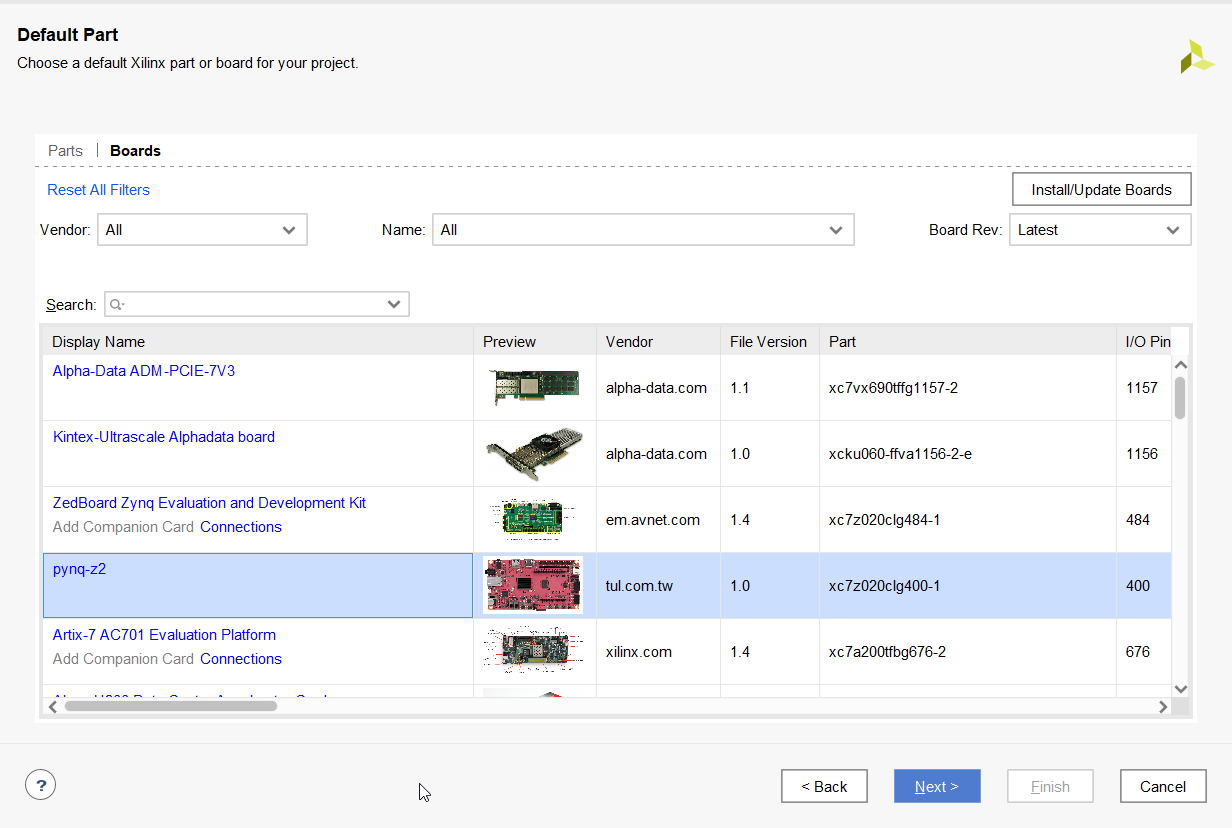
综上所述，项目可以完成题目指定任务

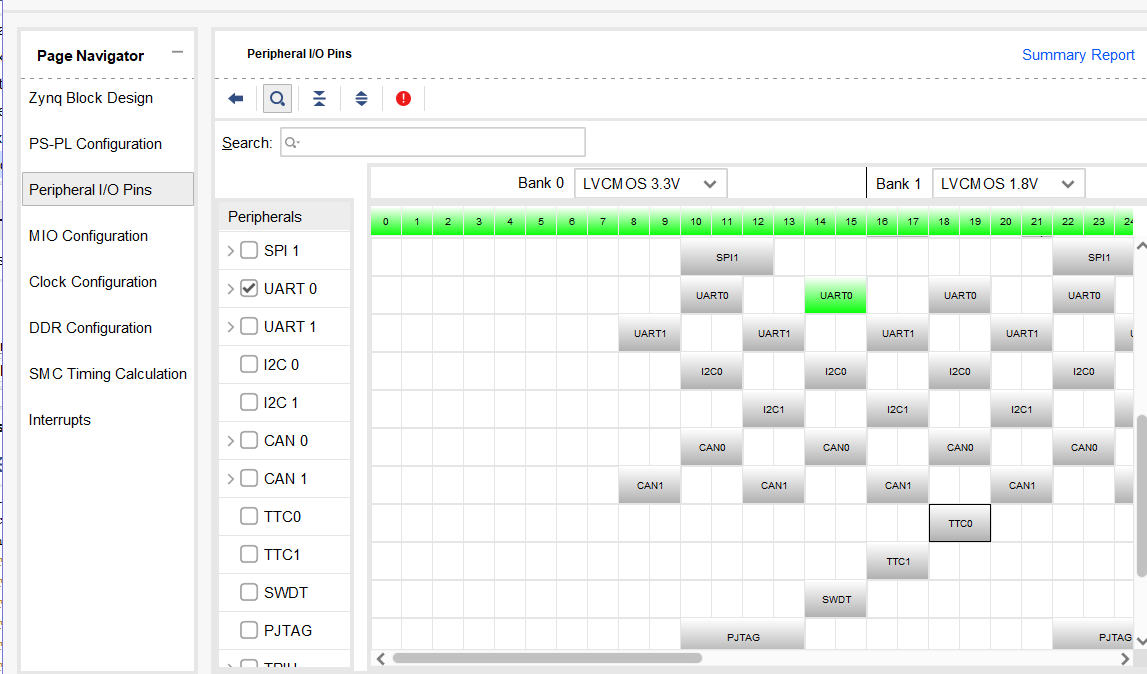
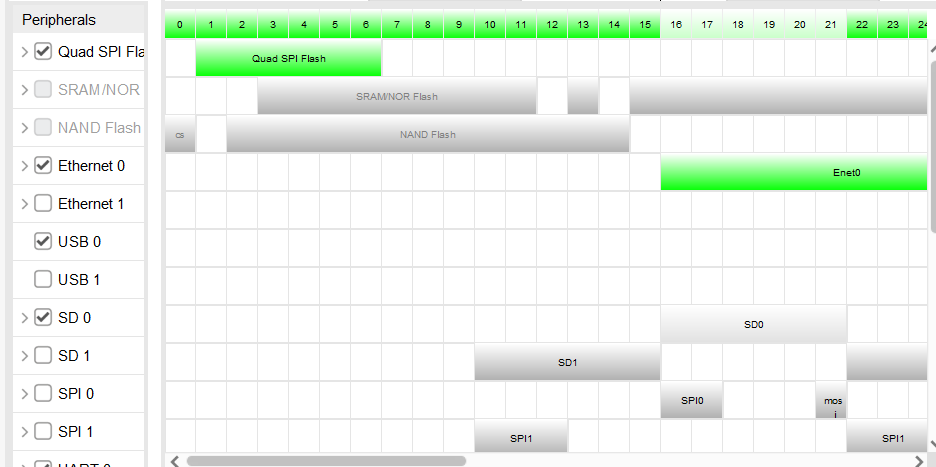
**【设计文件概述】**

本部分只介绍项目中文件安排和软硬件协同架构设计，具体实现在后续部分展示

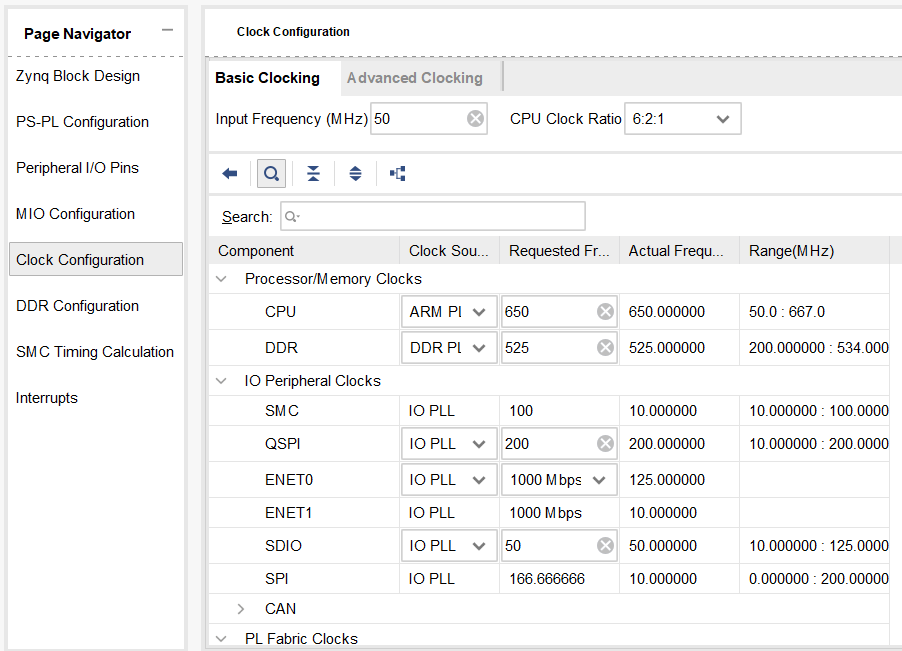
项目文件保存在同一目录，分为**Vivado封装IP**、**Vivado硬件设计**、**Vitis软件设计**三部分，大部分Verilog 编写RTL有注释，所有C文件和函数均经过规范注释，可通过Doxygen指令生成文档。

为方便外设驱动，这里使用了官方提供的板卡支持文件，不像之前实验直接配置约束，并直接使用了board file提供的物理约束和PS时序，关键配置如下图所示

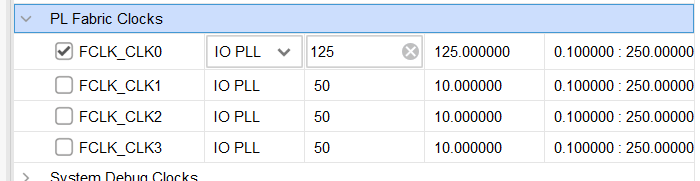




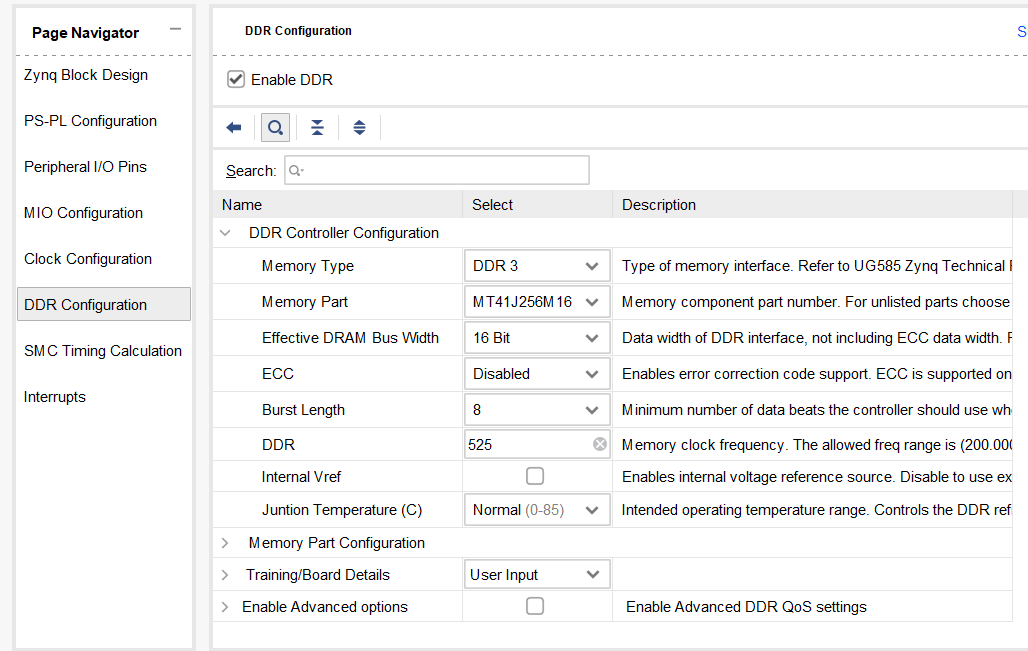
其中Cortex-A9采用了650MHz时钟，未跑满666MHz，DDR同样降频到了525MHz

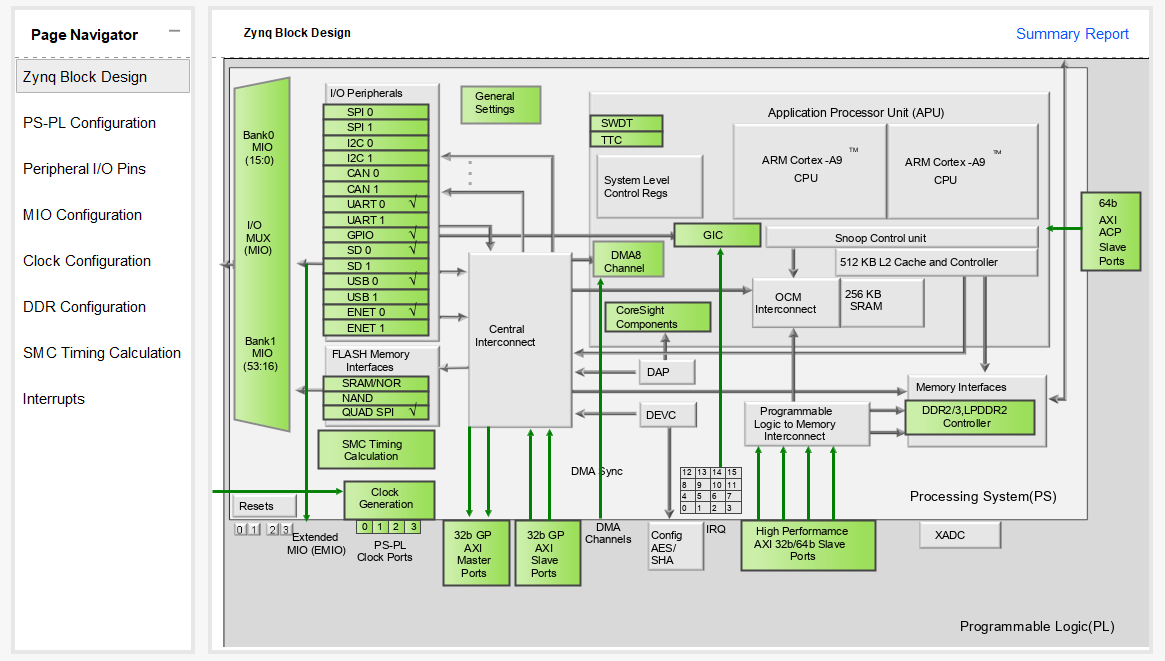


由于使用硬件实现RTC，没有开启TTC外设，同时将AXI-PL时钟拉高到125MHz以方便实验二中RTC IP的移植实现。



其他配置如下所示

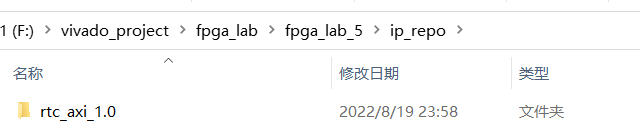
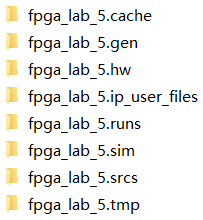




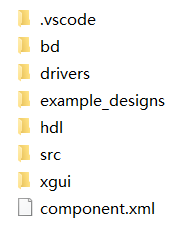
项目整体硬件设计如下：



工程中硬件部分使用到的RTL源文件皆为BlockDesign生成，Vivado文件保存至下列目录，自定义AXI-RTC文件保存至ip\_repo目录，如下所示

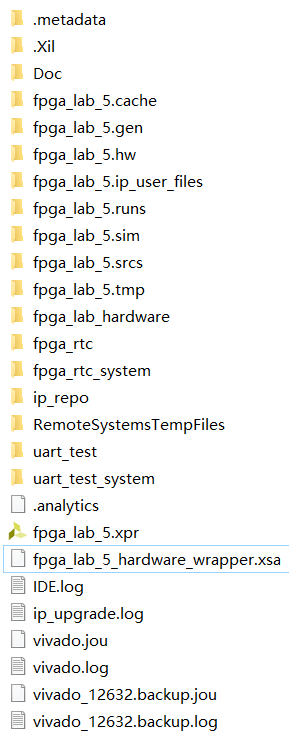


IP封装过程中使用到了实验二的RTL文件和TestBench文件，如下所示



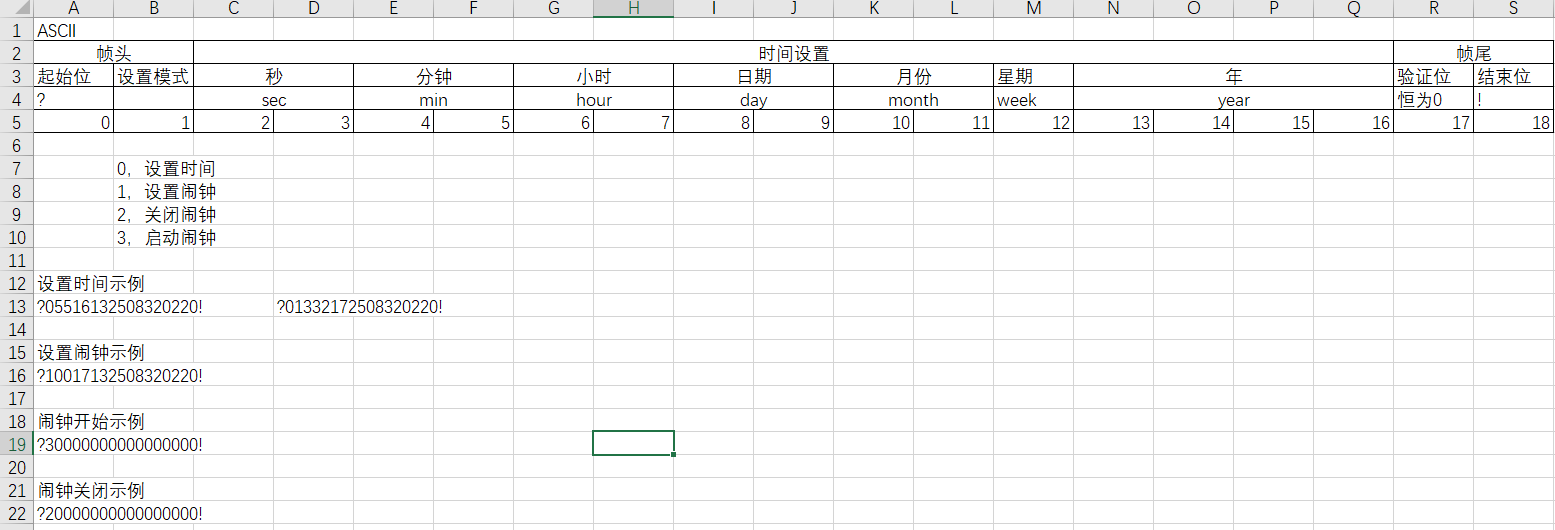
其中axi\_rtc\_top.v、alarm.v、alarm\_tb.v为后续添加，用以满足接驳AXI-Lite总线和闹钟功能。相关文件内容和设计思路在后续【设计文件与实验过程】部分展示。

硬件RTL代码综合布线生成比特流后导出硬件fpga\_lab\_5\_hardware\_wrapper.xsa，保存到工程根目录下，切换到Vitis进行嵌入式软件开发。文件系统目录和工程目录如下对应：



其中项目fpga\_lab\_hardware是Vitis的BSP工程，为上层软件提供基础的BootLoader和板级支持包，从xsa文件直接导出并配置了xilinx的baremetal裸金属库，实现了xil\_printf()串口打印功能和最基础的AXI总线寻址读写功能。项目fpga\_rtc是嵌入式软件部分的主要工程，项目在rtc.c、rtc.h中实现了AXI-RTC的HAL硬件抽象层功能，基于该库又使用display.c、display.h实现了rtc的操作功能，从而实现上下层软件解耦。主函数在application.c调用display.c中的三个TaskFunction，使用Zynq的内核SysTick硬件定时器实现非阻塞任务调度。项目uart\_test\_system则是串口示例程序的测试工程，方便切换到串口调试使用，与本实验主体项目实现无关，下面不再介绍。

串口控制部分使用到了自定义的数据帧结构，采用串口传输ASCII码的形式，更方便下位机的读取，帧结构如下



该文件定义在Doc目录下的ControlFormat.xlsx中

其中起始位为?，结束位为!，保留一个验证位写0，也可软件更换为奇偶校验或CRC8校验

当设置模式位为0时，表示后续数据用于设置时间

当设置模式位为1时，表示后续数据用于设置闹钟

当设置模式位为2时，忽略后续数据，关闭闹钟功能

当设置模式位为3时，忽略后续数据，开启闹钟功能

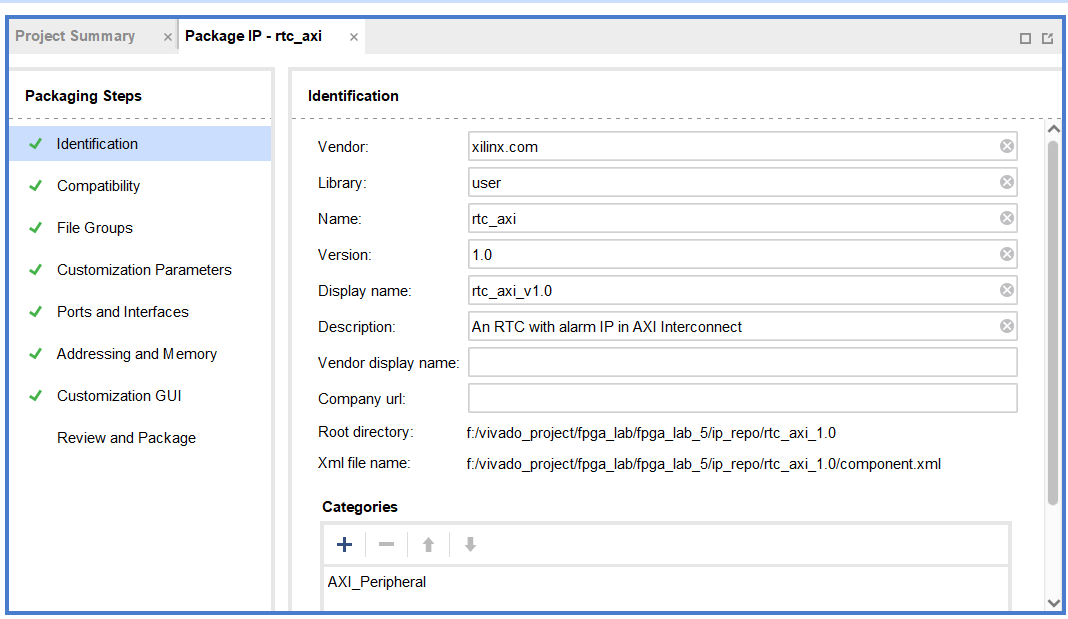
具体使用如演示视频所示

软硬件部分均使用VSCode作为代码编辑器，分别使用Vivado GUI进行硬件综合调试和调用XSCT（Xilinx Software Commandline Tool）进行软件生成编译以及最终烧录，因此在项目文件中存在.vscode目录

**【设计文件与实验过程】**

**IP封装**

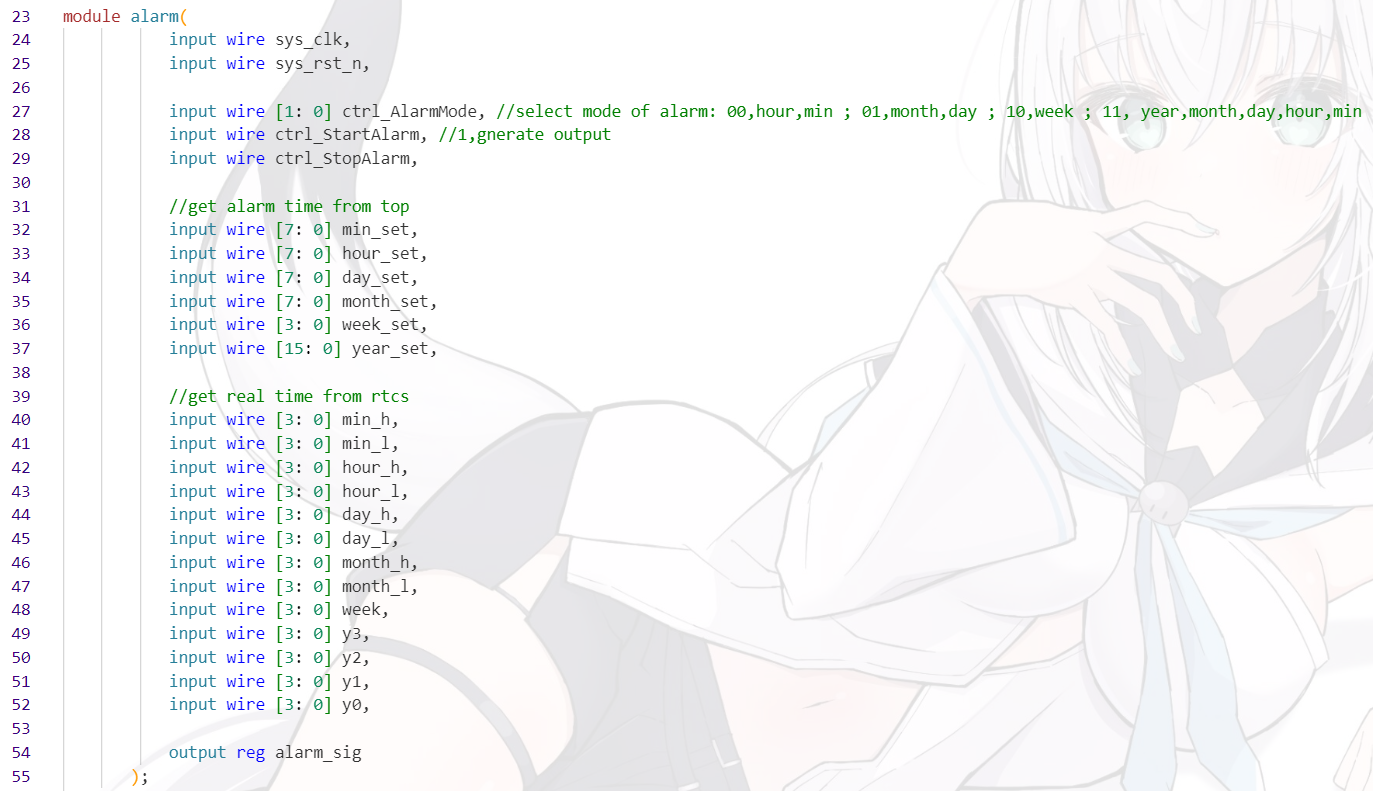
项目中使用了IP Packager在AXI封装模式下封装IP，得到结果如下图所示



IP使用到的源文件展示如下，工程目录保存在fpga\_lab\_5.tmp

【rtc.v】、【divider.v】、【leapyear.v】在实验二基础上没有改动

【alarm.v】



读取当前RTC输出时间，并与从顶层模块得到的闹钟时间作比较，如果当前时间等于闹钟时间，则挂起闹钟响铃信号

实现了三个控制寄存器：闹钟开启、闹钟停止、闹钟模式控制

闹钟模式控制寄存器可以控制用于判断的数据是（小时、分钟）或（月份、日期）或（星期）或（年月日时分星期）四种模式中的一种。

代码如下所示。

reg alarm\_en;

always @( \* ) begin

    case (ctrl\_AlarmMode)

        2'b00: begin //00,hour,min

            if (

                ({min\_h, min\_l} == min\_set) &&

                ({hour\_h, hour\_l} == hour\_set)

            ) begin

                alarm\_en = 1'b1;

            end

            else begin

                alarm\_en = 1'b0;

            end

        end

        2'b01: begin //01,month,day

            if (

                ({day\_h, day\_l} == day\_set) &&

                ({month\_h, month\_l} == month\_set)

            ) begin

                alarm\_en = 1'b1;

            end

            else begin

                alarm\_en = 1'b0;

            end

        end

        2'b10: begin //10,week

            if (week == week\_set) begin

                alarm\_en = 1'b1;

            end

            else begin

                alarm\_en = 1'b0;

            end

        end

        2'b11: begin //year,month,day,hour,min

            if (

                ({min\_h, min\_l} == min\_set) &&

                ({hour\_h, hour\_l} == hour\_set) &&

                ({day\_h, day\_l} == day\_set) &&

                ({month\_h, month\_l} == month\_set) &&

                ({y3, y2, y1, y0} == year\_set)

            ) begin

                alarm\_en = 1'b1;

            end

            else begin

                alarm\_en = 1'b0;

            end

        end

        default: begin

            alarm\_en = 1'b0;

        end

    endcase

end

always @(posedge sys\_clk or negedge sys\_rst\_n) begin

    if (!sys\_rst\_n) begin

        alarm\_sig <= 1'b0;

    end

    else if (ctrl\_StartAlarm) begin //control alarm-start

        if (alarm\_en) begin

            alarm\_sig <= 1'b1; //generate alarm signal

        end

        else begin

            alarm\_sig <= alarm\_sig;

        end

    end

    else if (ctrl\_StopAlarm) begin //if catch a stop signal, stop the alarm

        alarm\_sig <= 1'b0;

    end

    else begin

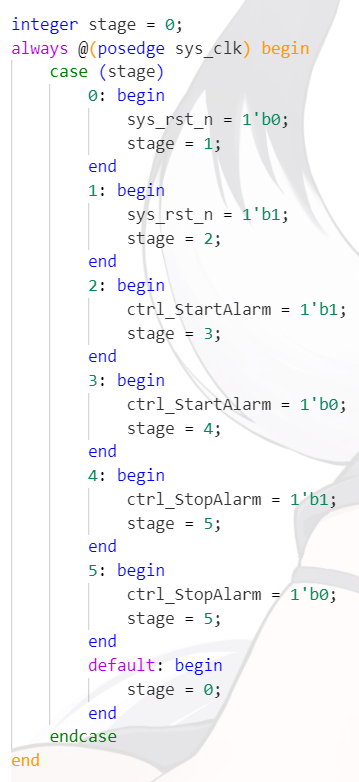
        alarm\_sig <= alarm\_sig;

    end

end

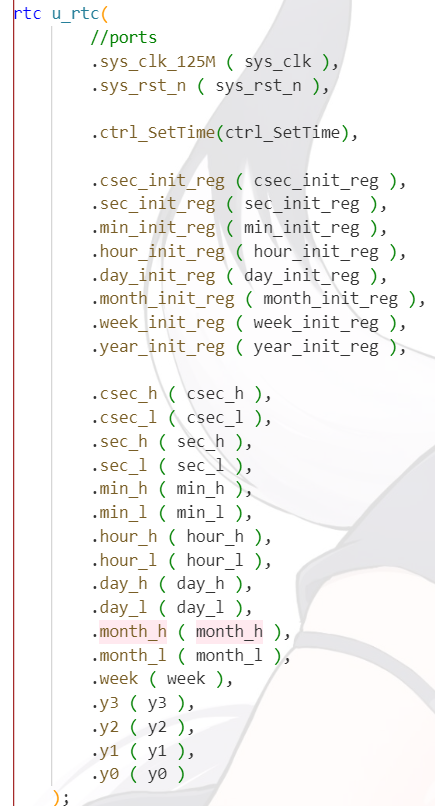
【alarm\_tb.v】

测试向量，用于判断硬件闹钟模块是否正常工作，关键代码如下所示



【axi\_rtc\_top.v】

带闹钟RTC的顶层模块，其中例化了alarm和rtc模块，使用两个多路选择器连线实现ctrl\_SetAlarm和的ctrl\_SetTime逻辑



控制逻辑如下所示：

always @( \* ) begin

    if (!sys\_rst\_n) begin

        csec\_init\_reg = 8'h00;

        sec\_init\_reg = 8'h00;

        min\_init\_reg = 8'h00;

        hour\_init\_reg = 8'h00;

        day\_init\_reg = 8'h01; //1st

        month\_init\_reg = 8'h01; //Jan

        week\_init\_reg = 4'h01; //Monday

        year\_init\_reg = 16'h2\_0\_2\_2; //2022

    end

    else if (ctrl\_SetTime) begin

        {csec\_init\_reg, sec\_init\_reg, min\_init\_reg, hour\_init\_reg, day\_init\_reg, month\_init\_reg, week\_init\_reg, year\_init\_reg} = rtc\_init\_reg;

    end

    else begin

        csec\_init\_reg = csec\_init\_reg;

        sec\_init\_reg = sec\_init\_reg;

        min\_init\_reg = min\_init\_reg;

        hour\_init\_reg = hour\_init\_reg;

        day\_init\_reg = day\_init\_reg;

        week\_init\_reg = week\_init\_reg;

        month\_init\_reg = month\_init\_reg;

        year\_init\_reg = year\_init\_reg;

    end

end

always @( \* ) begin

    if (!sys\_rst\_n) begin

        min\_set = 8'h00;

        hour\_set = 8'h00;

        day\_set = 8'h01; //1st

        month\_set = 8'h01; //Jan

        week\_set = 4'h01; //Monday

        year\_set = 16'h2\_0\_2\_2; //2022

    end

    else if (ctrl\_SetAlarm) begin

        {min\_set, hour\_set, day\_set, month\_set, week\_set, year\_set} = alarm\_init\_reg;

    end

    else begin

        min\_set = min\_set;

        hour\_set = hour\_set;

        day\_set = day\_set;

        month\_set = month\_set;

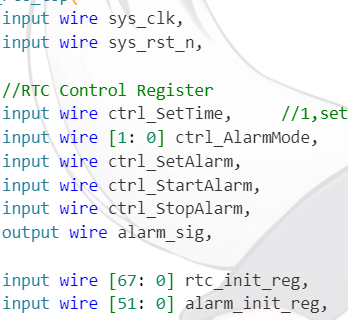
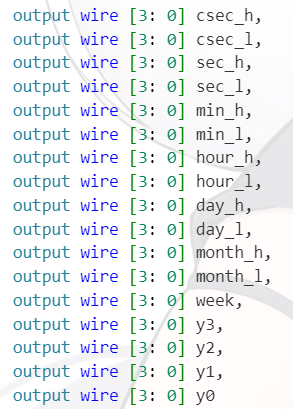
        week\_set = week\_set;

        year\_set = year\_set;

    end

end

模块引出了当前时间输出接口、闹钟信号输出接口、RTC配置接口（包含时间设置使能位、闹钟设置使能位、闹钟开启信号、闹钟关闭信号、闹钟模式选择位）以及系统时钟和复位信号接口



ctrl\_SetTime拉高会让RTC进入时间设置模式，设置信号rtc\_init\_reg有效

ctrl\_AlarmMode选择不同位会让闹钟进入不同工作模式

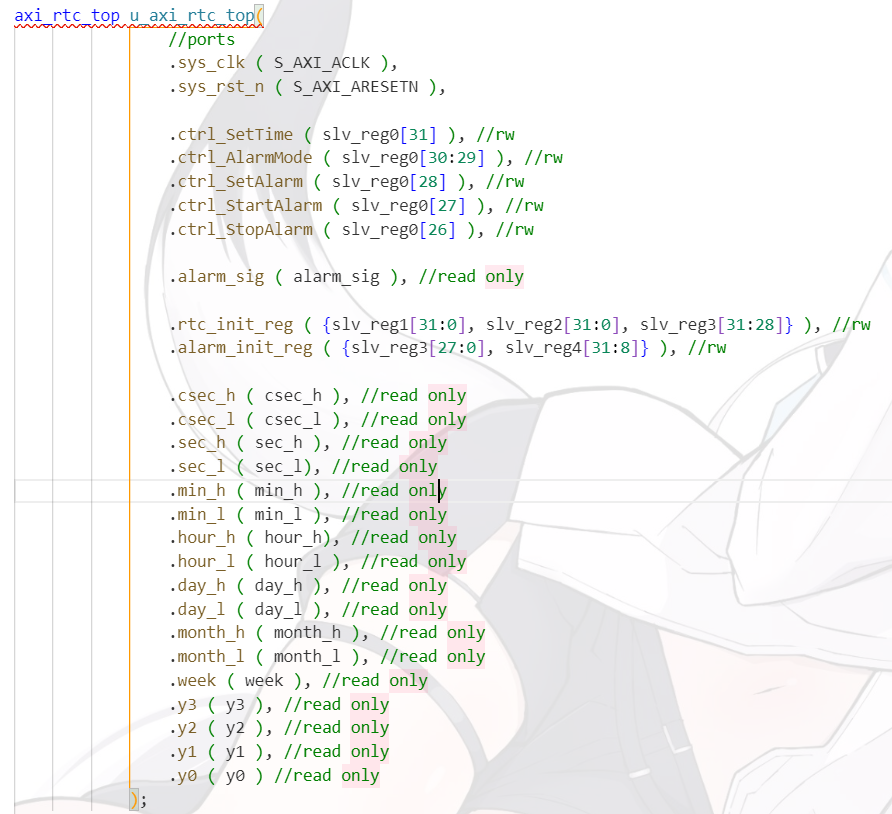
ctrl\_SetAlarm拉高会让闹钟进入设置模式，设置信号alarm\_init\_reg有效

ctrl\_StopAlarm拉高会停止闹钟运作

alarm\_sig用于闹钟信号输出，拉高表示闹铃响，拉低表示正常工作状态

【rtc\_axi\_v1\_0\_S00\_AXI.v】

AXI-Lite接口实现。在其中例化了RTC-ALARM顶层模块，并修改了部分AXI-Lite总线接口代码来实现只读和读写两种操作格式的寄存器



在其中加入了组合逻辑如下：

always @(\*) begin

    // slv\_reg0[25] = alarm\_sig;

    slv\_reg5[31:28] = csec\_h;

    slv\_reg5[27:24] = csec\_l;

    slv\_reg5[23:20] = sec\_h;

    slv\_reg5[19:16] = sec\_l;

    slv\_reg5[15:12] = min\_h;

    slv\_reg5[11:8] = min\_l;

    slv\_reg5[7:4] = hour\_h;

    slv\_reg5[3:0] = hour\_l;

    slv\_reg6[31:28] = day\_h;

    slv\_reg6[27:24] = day\_l;

    slv\_reg6[23:20] = month\_h;

    slv\_reg6[19:16] = month\_l;

    slv\_reg6[15:12] = week;

    slv\_reg6[11:8] = y3;

    slv\_reg6[7:4] = y2;

    slv\_reg6[3:0] = y1;

    slv\_reg7[31:28] = y0;

    slv\_reg7[0] = alarm\_sig;

end

用于实现只读的输出寄存器，以上寄存器（reg5、reg6、reg7）的**写操作**相关代码都被删除来屏蔽AXI主机的访问

修改后的代码如下所示：

always @( posedge S\_AXI\_ACLK ) begin

    if ( S\_AXI\_ARESETN == 1'b0 ) begin

        slv\_reg0 <= 0;

        slv\_reg1 <= 0;

        slv\_reg2 <= 0;

        slv\_reg3 <= 0;

        slv\_reg4 <= 0;

        //!disable write

        // slv\_reg5 <= 0;

        // slv\_reg6 <= 0;

        // slv\_reg7 <= 0;

    end

    else begin

        if (slv\_reg\_wren) begin

            case ( axi\_awaddr[ADDR\_LSB + OPT\_MEM\_ADDR\_BITS: ADDR\_LSB] )

                3'h0:

                    for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                        if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                            // Respective byte enables are asserted as per write strobes

                            // Slave register 0

                            slv\_reg0[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                        end

                3'h1:

                    for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                        if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                            // Respective byte enables are asserted as per write strobes

                            // Slave register 1

                            slv\_reg1[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                        end

                3'h2:

                    for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                        if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                            // Respective byte enables are asserted as per write strobes

                            // Slave register 2

                            slv\_reg2[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                        end

                3'h3:

                    for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                        if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                            // Respective byte enables are asserted as per write strobes

                            // Slave register 3

                            slv\_reg3[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                        end

                3'h4:

                    for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                        if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                            // Respective byte enables are asserted as per write strobes

                            // Slave register 4

                            slv\_reg4[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                        end

                //!disable read

                // 3'h5:

                //     for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                //         if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                //             // Respective byte enables are asserted as per write strobes

                //             // Slave register 5

                //             slv\_reg5[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                //         end

                // 3'h6:

                //     for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                //         if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                //             // Respective byte enables are asserted as per write strobes

                //             // Slave register 6

                //             slv\_reg6[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                //         end

                // 3'h7:

                //     for ( byte\_index = 0; byte\_index <= (C\_S\_AXI\_DATA\_WIDTH / 8) - 1; byte\_index = byte\_index + 1 )

                //         if ( S\_AXI\_WSTRB[byte\_index] == 1 ) begin

                //             // Respective byte enables are asserted as per write strobes

                //             // Slave register 7

                //             slv\_reg7[(byte\_index \* 8) +: 8] <= S\_AXI\_WDATA[(byte\_index \* 8) +: 8];

                //         end

                default : begin

                    slv\_reg0 <= slv\_reg0;

                    slv\_reg1 <= slv\_reg1;

                    slv\_reg2 <= slv\_reg2;

                    slv\_reg3 <= slv\_reg3;

                    slv\_reg4 <= slv\_reg4;

                    //!disable read

                    // slv\_reg5 <= slv\_reg5;

                    // slv\_reg6 <= slv\_reg6;

                    // slv\_reg7 <= slv\_reg7;

                end

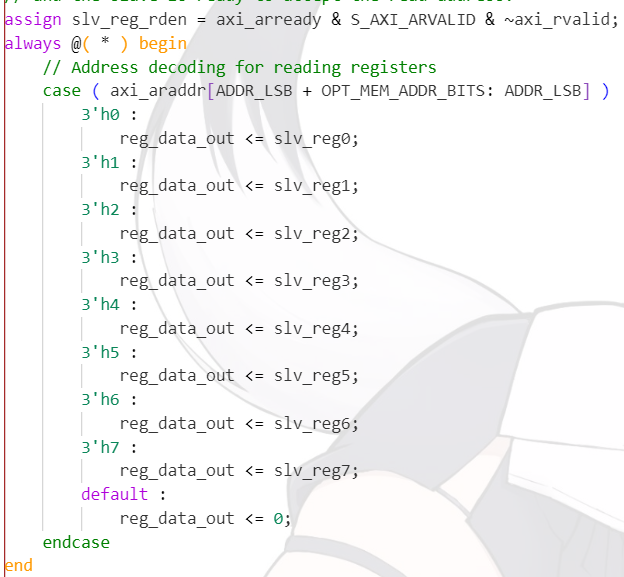
            endcase

        end

    end

end

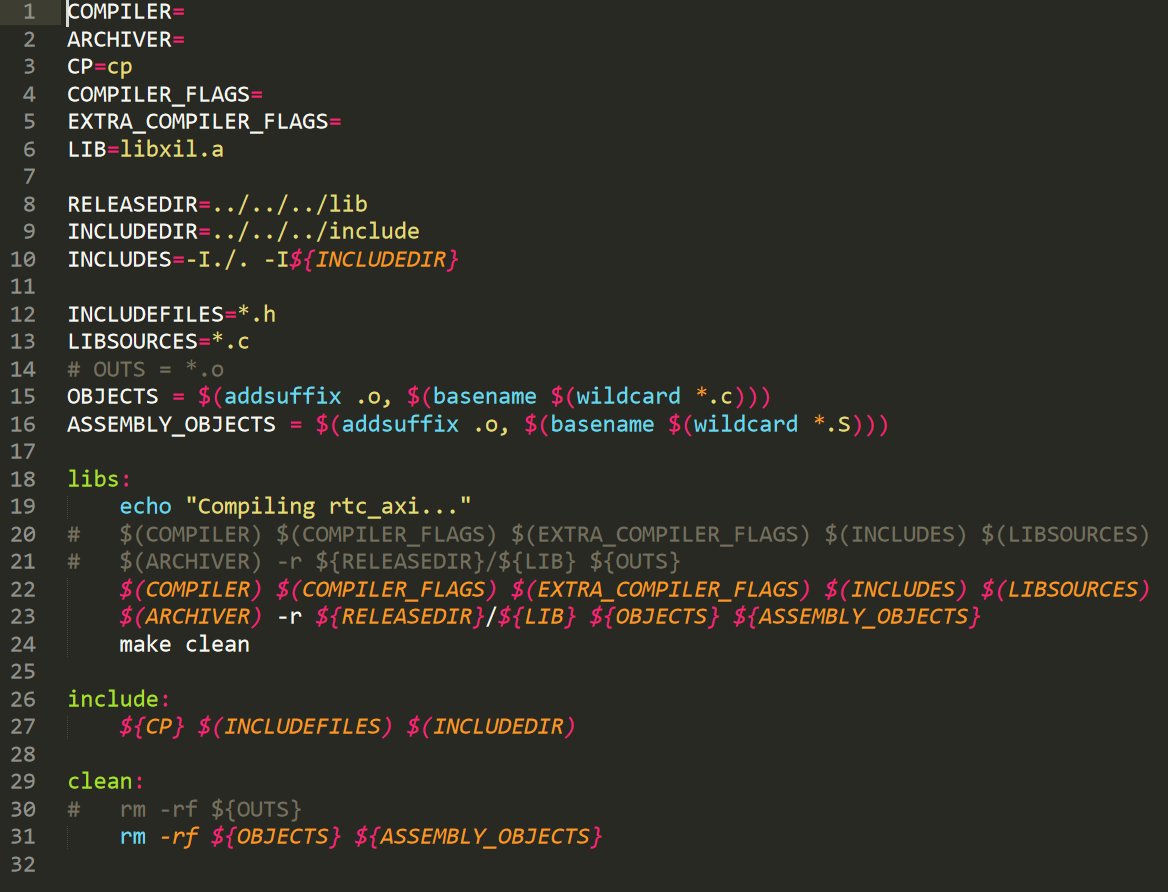
下面的取址代码没有被删除



【rtc\_axi\_v1\_0.v】

AXI-RTC IP的顶层模块，对下面的AXI-Lite控制器和RTC IP进行了一层标准AXI接口封装，用于和Xilinx的BlockDesign对接。使用以上文件完成RTL编写验证后进行封装，生成IP

对生成的IP改动如下：

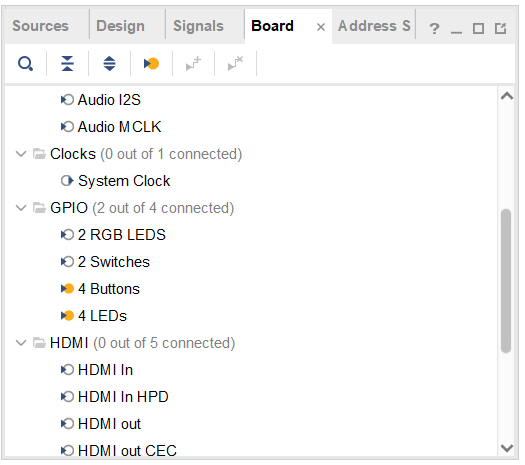
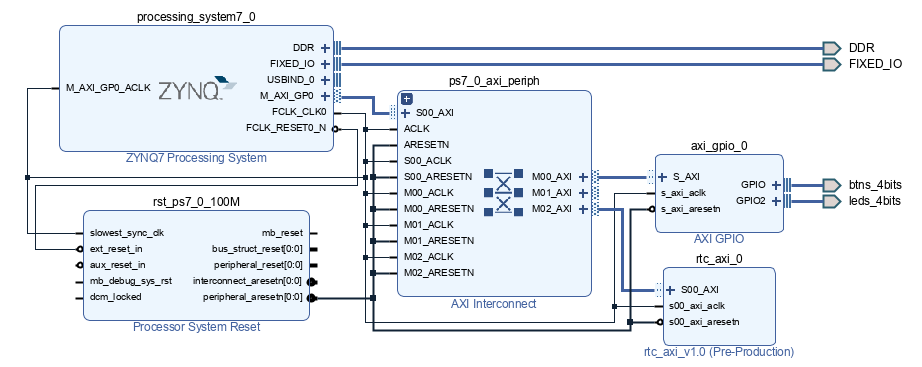


**硬件总体设计**

项目在上述IP和AXI GPIO IP核基础上，构建了基于ZYNQ PS部分的嵌入式系统，主要调用的PS硬件为串口，PL硬件为AXI-RTC和AXI-GPIO

整体互联结构如下，引入PSR IP作为系统复位控制器，使用系统慢速时钟作为IP核主时钟，由PS复位引脚作为其外部输入同步复位信号，向外设发送变换后的异步复位信号（相对IP核主时钟的异步复位）。

模块中还因此了八个控制引脚到外设LED和按钮，分别使用到了Board File提供的AXI-GPIO IP及对应物理约束



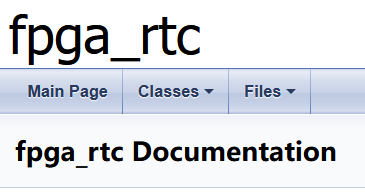
进行编译综合、优化布线、导出比特流、导出硬件文件，至此硬件设计部分结束，转入Vitis嵌入式软件开发阶段

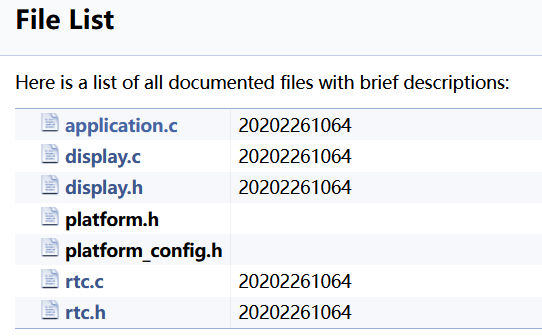
**软件总体设计**

软件设计中采用了**分层设计思想**和**C抽象数据类型（ADT）**，从而让应用代码和Xilinx提供的驱动风格较为统一，方便修改

下面用到了Doxygen工具来提供函数功能说明，Doxygen生成的html文件保存至fpga\_lab\_5\fpga\_rtc\src\DoxyOuts\html目录下，进入以后打开index.html即可查阅函数文档

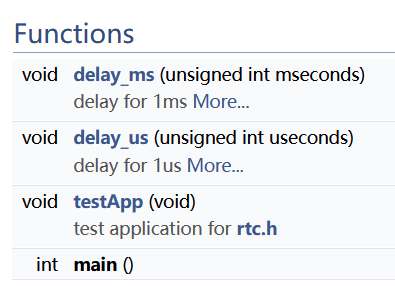
Doxyfile中只修改了PROJECT\_NAME属性



【application.c】

该文件是项目的主程序，包含如下主要函数：



void delay\_ms(unsigned int mseconds)

{

    XTime tEnd, tCur;

    XTime\_GetTime(&tCur);

    tEnd = tCur + (((XTime)mseconds) \* (COUNTS\_PER\_SECOND / 1000));

    do

    {

        XTime\_GetTime(&tCur);

    } while (tCur < tEnd);

}

void delay\_us(unsigned int useconds)

{

    XTime tEnd, tCur;

    XTime\_GetTime(&tCur);

    tEnd = tCur + (((XTime)useconds) \* (COUNTS\_PER\_SECOND / 1000000));

    do

    {

        XTime\_GetTime(&tCur);

    } while (tCur < tEnd);

}

这两个函数提供基于内核SysTick定时器的阻塞形式的ms和us延迟，由于其阻塞特性故没有在代码中正式使用。

void testApp(void)

该函数提供了RTC的输出测试，在正式代码中未采用。

项目中用到的主函数如下所示：

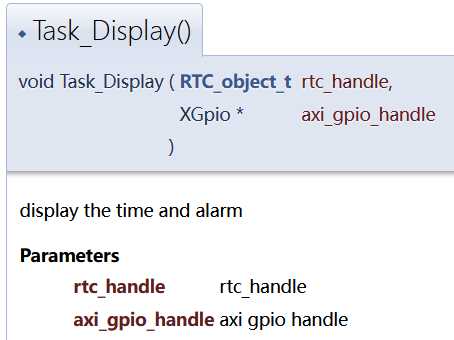


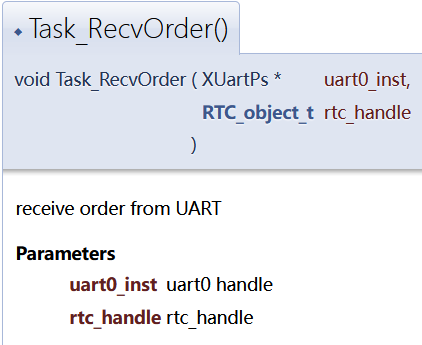
函数的最开始部分进行BSP初始化，调用了display.c中提供的初始化API，并在系统初始化结束后打印System Start

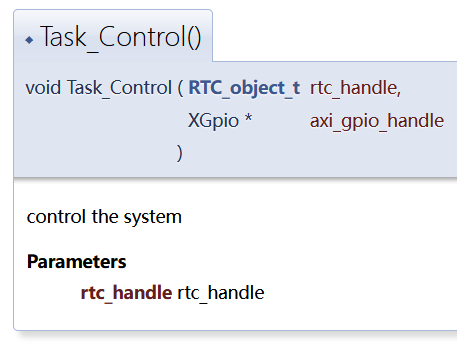
主循环之前进行非阻塞延时的初始化，后面的Task\_Display()函数基于非阻塞的硬件定时器延时来提高运行效率

最终的cleanup\_platform()并不会执行，出于代码风格统一而保留

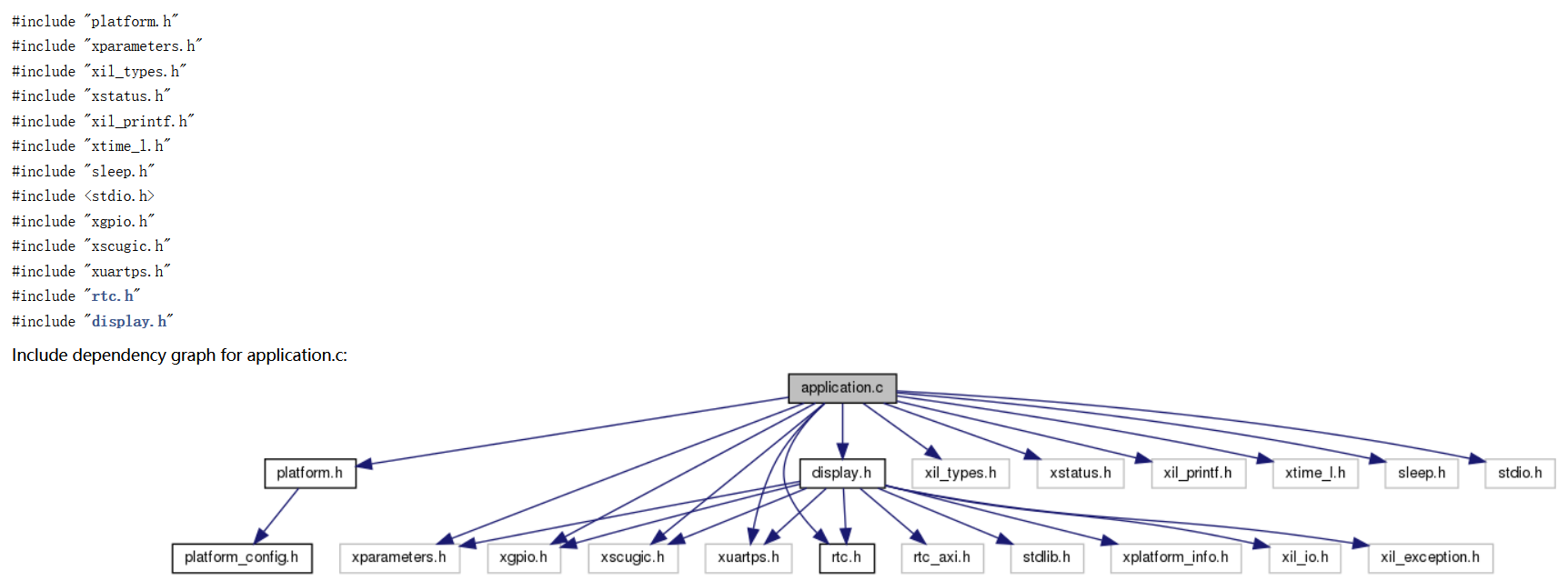
主循环执行三个任务：显示、接收数据和控制硬件，功能描述如下所示：







该文件的库依赖如下所示：

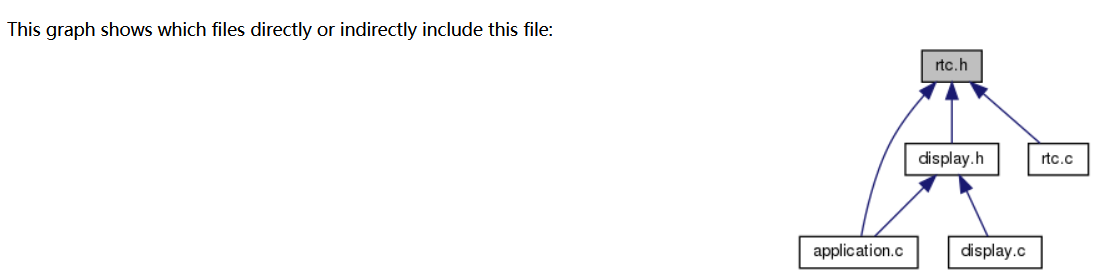


可以发现只用到了硬件相关和Xilinx的定义库，耦合度相对较低

【rtc.h】【rtc.c】

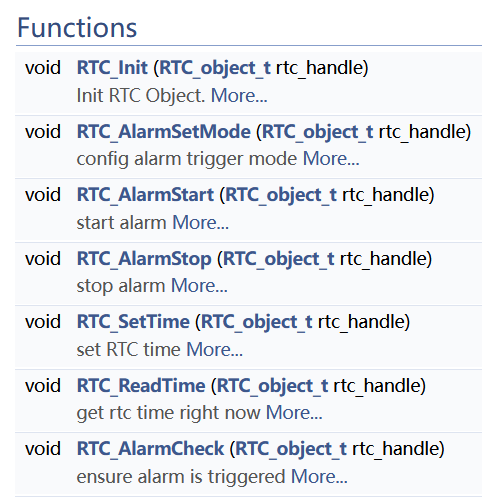
项目中用到了RTC的驱动函数，但Vitis生成BSP时只提供了基础的寄存器IO驱动，封装IP时也没有对BSP驱动进行编写，因此在这里加入了一层可以移植的位于HAL硬件抽象层的硬件驱动来方便上层设备调用





从库依赖图可以看到本HAL驱动只使用到了Vitis生成的rtc\_axi.h和xil\_io.h，但其被所有顶层模块所依赖

AXI-RTC驱动提供了如下API：

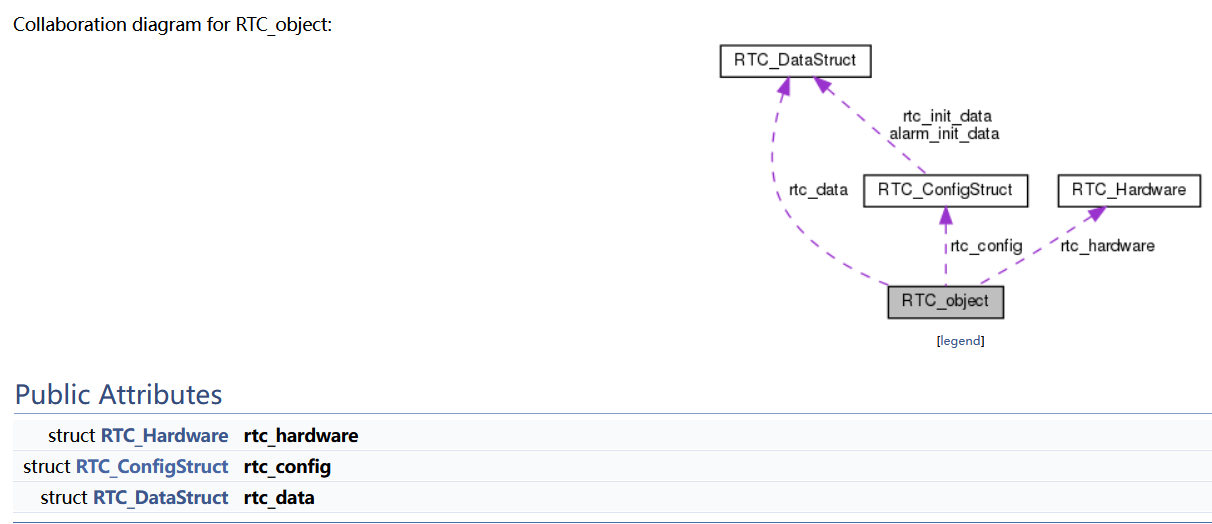


从上到下分别是：

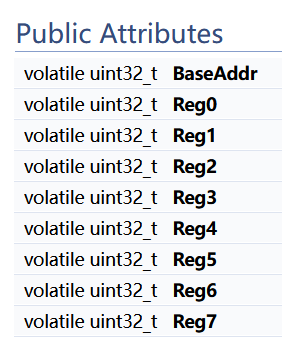
初始化RTC、设置闹钟触发模式、开启闹钟、关闭闹钟、设置实时时间和闹钟时间、读取当前实时时间、检查闹钟响铃信号是否触发。函数全部使用到了在头文件定义的RTC类。

本驱动采用C抽象数据类型（CADT）和C面向对象的设计，首先根据RTC的属性配置了RTC\_object\_t结构体指针数据类型，这是对RTC\_object类的封装（指针引用）。RTC\_object是对RTC硬件和软件配置的抽象类，包含如下属性：

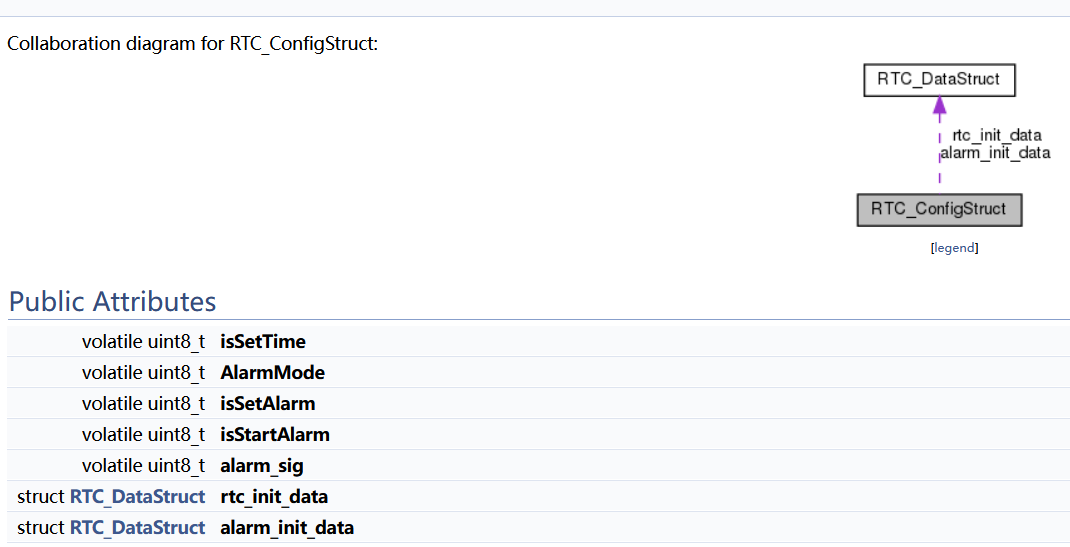
RTC硬件描述结构、RTC配置描述结构、RTC显示数据



RTC硬件描述结构包含了所有的RTC寄存器地址，并可以在RTC\_Init()函数中进行初始化，是对RTC硬件的抽象



RTC配置描述结构如下所示：



包含了所有RTC要用到的配置属性，包括RTC设置时间和闹钟设置时间

RTC\_object类中所有的数据，包括配置数据和提供给API的RTC显示数据都使用了RTC\_DataStruct作为数据结构，它包含了百分秒、秒、分钟、小时、天、月份、星期、年，**本驱动下所有操作都是对该数据结构行读写**



完成数据结构设计，下面介绍算法主体：

为了风格统一，全部public函数的参数都只采用类句柄引用，私有函数则被封装在文件内部，包含以下几个：

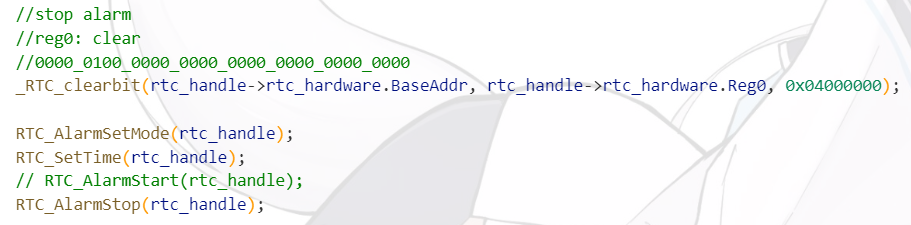


函数功能在注释中写出，主要实现了算法转换和外设寄存器按掩码读写的功能，方便其他API调用

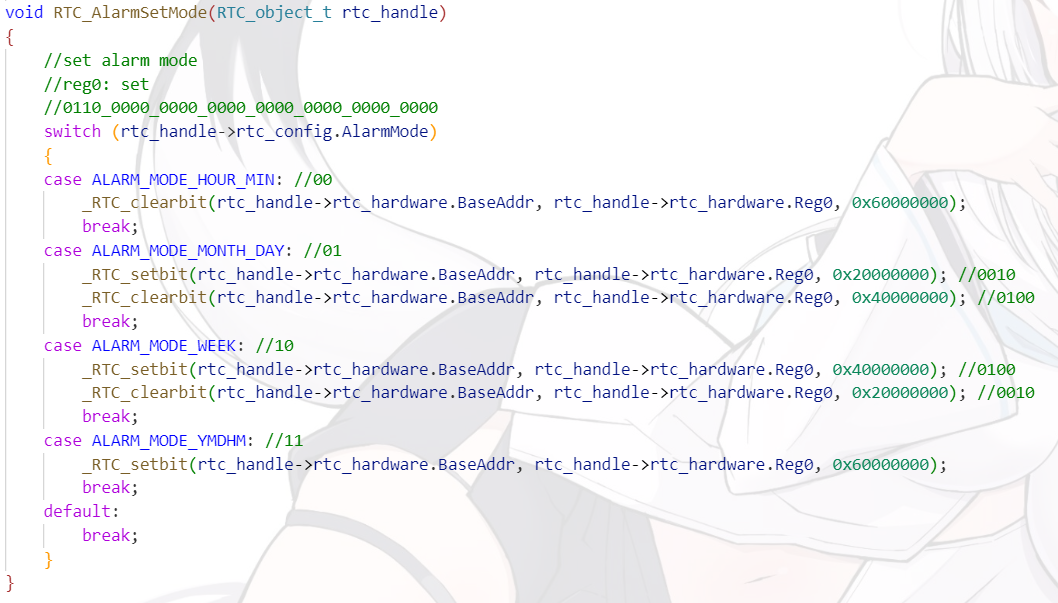
RTC\_Init函数为所有成员变量赋合适初值



同时保证关断闹钟以及设置起始时间



RTC\_AlarmSetMode根据模式选择属性的不同选择闹钟匹配模式



下面是开启闹钟和停止闹钟的实现



设置代码比较繁琐，摘录如下。主要思想是将8位或16位的C变量数据（保存在内存，以小端32位格式对齐）先扩展成32位计算出各位BCD值（4位）后通过移位拼接成到对应寄存器上

set\_time\_reg1 = ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.csec / 10) << 28) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.csec % 10) << 24) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.sec / 10) << 20) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.sec % 10) << 16) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.min / 10) << 12) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.min % 10) << 8) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.hour / 10) << 4) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.hour % 10));

        set\_time\_reg2 = ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.day / 10) << 28) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.day % 10) << 24) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.month / 10) << 20) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.month % 10) << 16) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.week) << 12) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.year / 1000) << 8) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.year % 1000 / 100) << 4) |

            ((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.year % 100 / 10));

        set\_time\_reg3 = (((uint32\_t)(rtc\_handle->rtc\_config.rtc\_init\_data.year % 10)) << 28);

        \_RTC\_setbit(rtc\_handle->rtc\_hardware.BaseAddr, rtc\_handle->rtc\_hardware.Reg3, set\_time\_reg3);

具体设置的是RTC还是闹钟则由控制成员变量isSetAlarm和isSetTime决定

读取RTC时间正好是其反操作，代码如下

void RTC\_ReadTime(RTC\_object\_t rtc\_handle)

{

    volatile uint32\_t reg5 = 0;

    volatile uint32\_t reg6 = 0;

    volatile uint32\_t reg7 = 0;

    reg5 = RTC\_AXI\_mReadReg(rtc\_handle->rtc\_hardware.BaseAddr, rtc\_handle->rtc\_hardware.Reg5);

    reg6 = RTC\_AXI\_mReadReg(rtc\_handle->rtc\_hardware.BaseAddr, rtc\_handle->rtc\_hardware.Reg6);

    reg7 = RTC\_AXI\_mReadReg(rtc\_handle->rtc\_hardware.BaseAddr, rtc\_handle->rtc\_hardware.Reg7);

    rtc\_handle->rtc\_data.csec = \_RTC\_bcd2dec((reg5 & 0xF0000000) >> 24, (reg5 & 0x0F000000) >> 24);

    rtc\_handle->rtc\_data.sec = \_RTC\_bcd2dec((reg5 & 0x00F00000) >> 16, (reg5 & 0x000F0000) >> 16);

    rtc\_handle->rtc\_data.min = \_RTC\_bcd2dec((reg5 & 0x0000F000) >> 8, (reg5 & 0x00000F00) >> 8);

    rtc\_handle->rtc\_data.hour = \_RTC\_bcd2dec((reg5 & 0x000000F0), (reg5 & 0x0000000F));

    rtc\_handle->rtc\_data.day = \_RTC\_bcd2dec((reg6 & 0xF0000000) >> 24, (reg6 & 0x0F000000) >> 24);

    rtc\_handle->rtc\_data.month = \_RTC\_bcd2dec((reg6 & 0x00F00000) >> 16, (reg6 & 0x000F0000) >> 16);

    rtc\_handle->rtc\_data.week = (reg6 & 0x0000F000) >> 12;

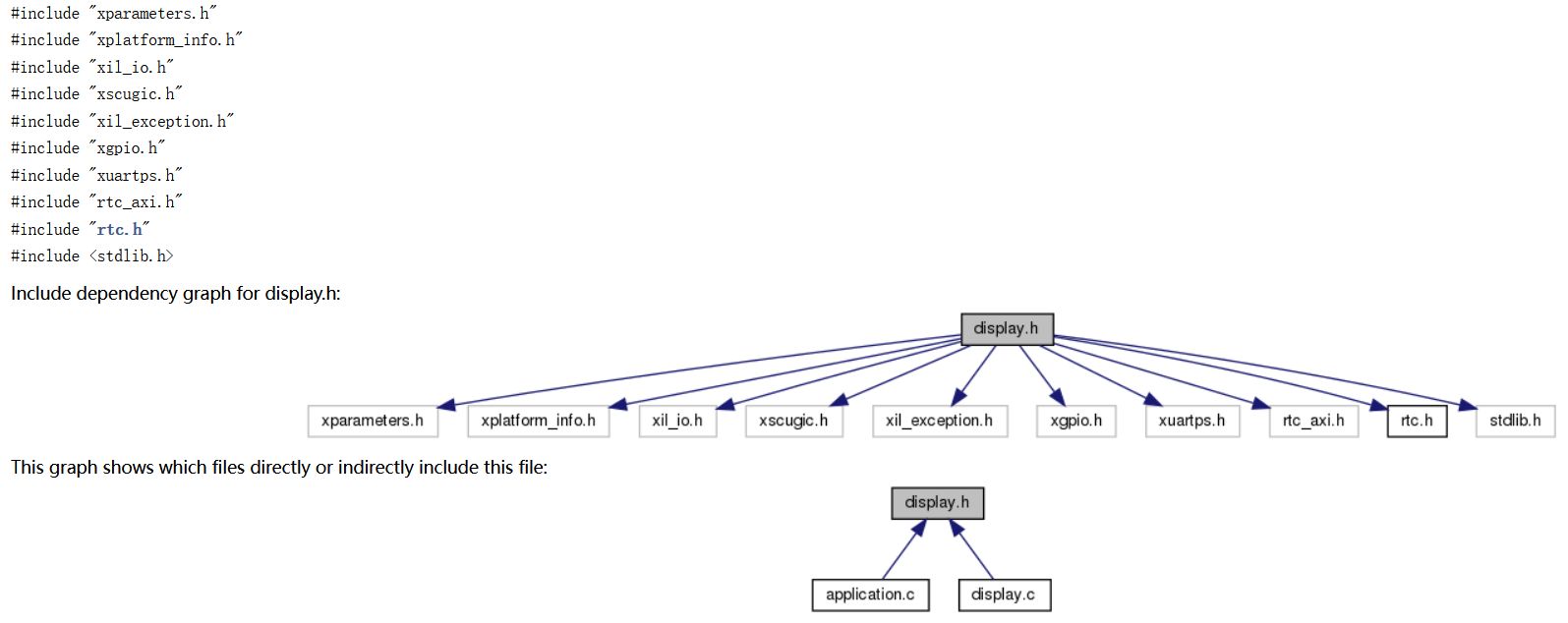
    rtc\_handle->rtc\_data.year = ((reg6 & 0x00000F00) >> 8) \* 1000 + ((reg6 & 0x000000F0) >> 4) \* 100 + (reg6 & 0x0000000F) \* 10 + ((reg7 & 0xF0000000) >> 28);

}

这里使用到之前编写的内部成员函数\_RTC\_bcd2dec()来实现4位BCD码到8位或16位十进制的转换

RTC驱动介绍到此结束，下面是应用层API介绍

【display.c】【display.h】



依赖如上所述，使用到了Xilinx提供的AXI-GPIO、UARTPS、GIC以及自行编写的RTC驱动

定义了如下宏和枚举类型来方便调用



该文件中定义的API就是在main函数中所使用到的主要API



全部使用到了与Xilinx驱动风格相同的指针引用，函数功能都已经写明

除此之外还定义了两个私有函数：

/\*\*

 \* @brief trans string value to dec number value

 \* @param  str              string

 \* @return uint16\_t dec number

 \*/

static uint16\_t string2dec(uint8\_t str)

{

    return str - '0';

}

/\*\*

 \* @brief calculate week from date

 \* @param  year             year num

 \* @param  month            month num

 \* @param  day              day num

 \* @return uint8\_t week

 \*/

static uint8\_t calculate\_week(uint16\_t year, uint8\_t month, uint8\_t day)

{

    uint8\_t century = year / 100;

    return ((uint32\_t)day + (2 \* (uint32\_t)month) + ((3 \* ((uint32\_t)month + 1)) / 5) + (uint32\_t)year + ((uint32\_t)year / 4) - ((uint32\_t)year / 100) + ((uint32\_t)year / 400)) % 7;

}

分别用于处理字符串（这里用到的字符串只有数字）转十进制和根据输入年月日检查星期是否与对应日期相符，计算星期用到了蔡勒公式

初始化函数都根据Xilinx提供的示例和官方文档**UG585**、**UG821**、**UG1165**编写，串口处注册了串口rxFIFO满中断，对应GIC中注册了串口中断服务回调函数







所有串口接收工作都在串口中断服务函数中进行，代码如下所示。

首先读中断配置寄存器判断中断类型，根据掩码检查是否为FIFO接收中断，完成中断处理函数后将中断标志位清零（即ARM Cortex系列的标准中断处理流程——读寄存器、检查标志位、处理中断、标志位清零、恢复现场）

中断中主要完成了接收该轮中断时FIFO中数据，将数据转存到内存缓存区的任务。并且如果中断触发后前几轮中断接收总数据量已经等于或超过了内存缓存区的数据量，说明一轮指令已经读完，可以等待下一轮数据，于是在此复位内存缓存区数据指针，并将总数据量计数器清零

/\*\*

 \* @brief UART0 Interrupt Request Handler Function

 \* @param  CallBackRef      UART0 inst passed from parent-function

 \*/

void UART0\_IRQHandler(void\* CallBackRef)

{

    XUartPs\* UartInstancePtr = (XUartPs\*)CallBackRef;

    uint32\_t IsrStatus;

    uint32\_t ReceivedCount = 0;

    //read interrupt reg to consider kinds of interrupt

    IsrStatus = XUartPs\_ReadReg(UartInstancePtr->Config.BaseAddress,

        XUARTPS\_IMR\_OFFSET);

    IsrStatus &= XUartPs\_ReadReg(UartInstancePtr->Config.BaseAddress,

        XUARTPS\_ISR\_OFFSET);

    //check if receiver FIFO trigger

    if (IsrStatus & (uint32\_t)XUARTPS\_IXR\_RXOVR)

    {

        ReceivedCount = XUartPs\_Recv(UartInstancePtr, RecvBufferPtr,

            BUFFER\_SIZE - TotalReceivedCount); //recv data from UART0 rxFIFO

        TotalReceivedCount += ReceivedCount; //move total value and buffer pointer

        RecvBufferPtr += ReceivedCount;

        XUartPs\_WriteReg(UartInstancePtr->Config.BaseAddress,

            XUARTPS\_ISR\_OFFSET, XUARTPS\_IXR\_RXOVR); //clear int flag

    }

    //idle interrupt trigger

//  else if (IsrStatus & (uint32\_t) XUARTPS\_IXR\_TOUT)

//  {

//      ReceivedCount = XUartPs\_Recv(UartInstancePtr, RecvBufferPtr,

//              (BUFFER\_SIZE - ReceivedCount));

//      XUartPs\_WriteReg(UartInstancePtr->Config.BaseAddress,

//              XUARTPS\_ISR\_OFFSET, XUARTPS\_IXR\_TOUT); //clear int flag

//  }

    if (TotalReceivedCount >= BUFFER\_SIZE) //compare total recv with Buffer length

    {

        //      xil\_printf("%s", RecvBuffer);

        //      xil\_printf("\r\nI have received %d bytes.\r\n", TotalReceivedCount);

        RecvBufferPtr = RecvBuffer; //if total recv > expect recv, enter next cycle to re-receive

        TotalReceivedCount = 0;

        UART\_RECV\_DONE = 1;

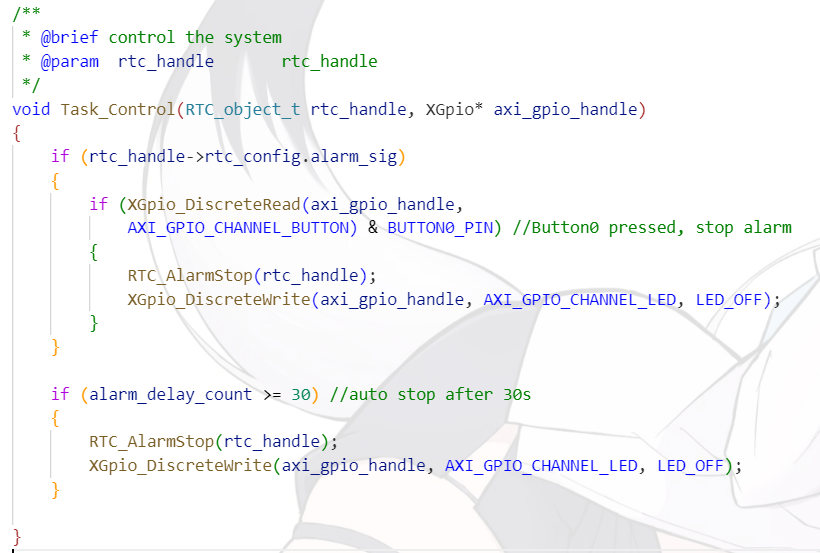
    }

}



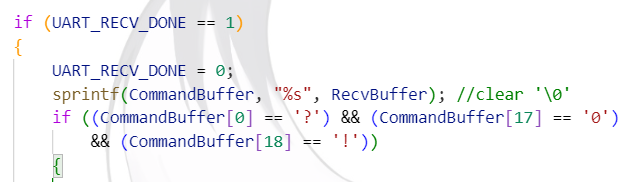
Task\_Display()函数分为三个主要步骤：读取RTC、检查是否有闹钟触发、输出当前时间

其中使用了一个转换逻辑将数字形式的星期值转换为字符串形式的英文缩写星期



Task\_Control()函数负责按钮和LED的控制，包括闹钟启动、按钮关闭闹钟和半分钟自停功能

Task\_RecvOrder()函数只有当每轮控制指令完成接收以后才会被调用到



UART\_RECV\_DONE标志位会在上面的串口中断函数中被置1，起到任务间通信（信号量）作用。sprintf语句用于消除内存缓存区末尾可能存在的’\0’

项目使用判断起始帧和结束帧结构的方式来确保指令格式正确，在确定接收串口数据是指令的情况下会输出Got Command信息。



Task\_RecvOrder()后续代码如下所示

if ((CommandBuffer[0] == '?') && (CommandBuffer[17] == '0')

            && (CommandBuffer[18] == '!'))

        {

            xil\_printf("Got Command!\r\n");

            switch (RecvBuffer[1])

            {

            case SET\_TIME: //'0'

                rtc\_handle->rtc\_config.isSetTime = 1;

                rtc\_handle->rtc\_config.rtc\_init\_data.csec = 0;

                rtc\_handle->rtc\_config.rtc\_init\_data.sec = string2dec(

                    RecvBuffer[2]) \* 10 + string2dec(RecvBuffer[3]);

                rtc\_handle->rtc\_config.rtc\_init\_data.min = string2dec(

                    RecvBuffer[4]) \* 10 + string2dec(RecvBuffer[5]);

                rtc\_handle->rtc\_config.rtc\_init\_data.hour = string2dec(

                    RecvBuffer[6]) \* 10 + string2dec(RecvBuffer[7]);

                rtc\_handle->rtc\_config.rtc\_init\_data.day = string2dec(

                    RecvBuffer[8]) \* 10 + string2dec(RecvBuffer[9]);

                rtc\_handle->rtc\_config.rtc\_init\_data.month = string2dec(

                    RecvBuffer[10]) \* 10 + string2dec(RecvBuffer[11]);

                rtc\_handle->rtc\_config.rtc\_init\_data.year = string2dec(

                    RecvBuffer[13]) \* 1000

                    + string2dec(RecvBuffer[14]) \* 100

                    + string2dec(RecvBuffer[15]) \* 10

                    + string2dec(RecvBuffer[16]);

                cal\_week = calculate\_week(rtc\_handle->rtc\_config.rtc\_init\_data.year, rtc\_handle->rtc\_config.rtc\_init\_data.month, rtc\_handle->rtc\_config.rtc\_init\_data.day);

                possible\_week = string2dec(RecvBuffer[12]);

                if (possible\_week == cal\_week)

                {

                    rtc\_handle->rtc\_config.rtc\_init\_data.week = possible\_week;

                }

                else

                {

                    xil\_printf("Error Week Input!\r\n");

                    rtc\_handle->rtc\_config.rtc\_init\_data.week = cal\_week;

                }

                RTC\_SetTime(rtc\_handle);

                xil\_printf("TIME SET: %d-%d-%d %d %d:%d:%d\r\n",

                    rtc\_handle->rtc\_config.rtc\_init\_data.year,

                    rtc\_handle->rtc\_config.rtc\_init\_data.month,

                    rtc\_handle->rtc\_config.rtc\_init\_data.day,

                    rtc\_handle->rtc\_config.rtc\_init\_data.week,

                    rtc\_handle->rtc\_config.rtc\_init\_data.hour,

                    rtc\_handle->rtc\_config.rtc\_init\_data.min,

                    rtc\_handle->rtc\_config.rtc\_init\_data.sec);

                break;

            case SET\_ALARM: //'1'

                rtc\_handle->rtc\_config.isSetAlarm = 1;

                rtc\_handle->rtc\_config.alarm\_init\_data.min = string2dec(

                    RecvBuffer[4]) \* 10 + string2dec(RecvBuffer[5]);

                rtc\_handle->rtc\_config.alarm\_init\_data.hour = string2dec(

                    RecvBuffer[6]) \* 10 + string2dec(RecvBuffer[7]);

                rtc\_handle->rtc\_config.alarm\_init\_data.day = string2dec(

                    RecvBuffer[8]) \* 10 + string2dec(RecvBuffer[9]);

                rtc\_handle->rtc\_config.alarm\_init\_data.month = string2dec(

                    RecvBuffer[10]) \* 10 + string2dec(RecvBuffer[11]);

                rtc\_handle->rtc\_config.alarm\_init\_data.year = string2dec(

                    RecvBuffer[13]) \* 1000

                    + string2dec(RecvBuffer[14]) \* 100

                    + string2dec(RecvBuffer[15]) \* 10

                    + string2dec(RecvBuffer[16]);

                possible\_week = string2dec(RecvBuffer[12]);

                cal\_week = calculate\_week(rtc\_handle->rtc\_config.alarm\_init\_data.year, rtc\_handle->rtc\_config.alarm\_init\_data.month, rtc\_handle->rtc\_config.alarm\_init\_data.day);

                if (possible\_week == cal\_week)

                {

                    rtc\_handle->rtc\_config.alarm\_init\_data.week = possible\_week;

                }

                else

                {

                    xil\_printf("Error Week Input!\r\n");

                    rtc\_handle->rtc\_config.alarm\_init\_data.week = cal\_week;

                }

                RTC\_SetTime(rtc\_handle);

                xil\_printf("ALARM SET: %d-%d-%d %d %d:%d\r\n",

                    rtc\_handle->rtc\_config.alarm\_init\_data.year,

                    rtc\_handle->rtc\_config.alarm\_init\_data.month,

                    rtc\_handle->rtc\_config.alarm\_init\_data.day,

                    rtc\_handle->rtc\_config.alarm\_init\_data.week,

                    rtc\_handle->rtc\_config.alarm\_init\_data.hour,

                    rtc\_handle->rtc\_config.alarm\_init\_data.min);

                break;

            case DISABLE\_ALRM: //'2'

                RTC\_AlarmStop(rtc\_handle);

                xil\_printf("Alarm Stop!\r\n");

                break;

            case ENABLE\_ALRM: //'3'

                RTC\_AlarmStart(rtc\_handle);

                xil\_printf("Alarm Begin!\r\n");

                break;

            default:

                break;

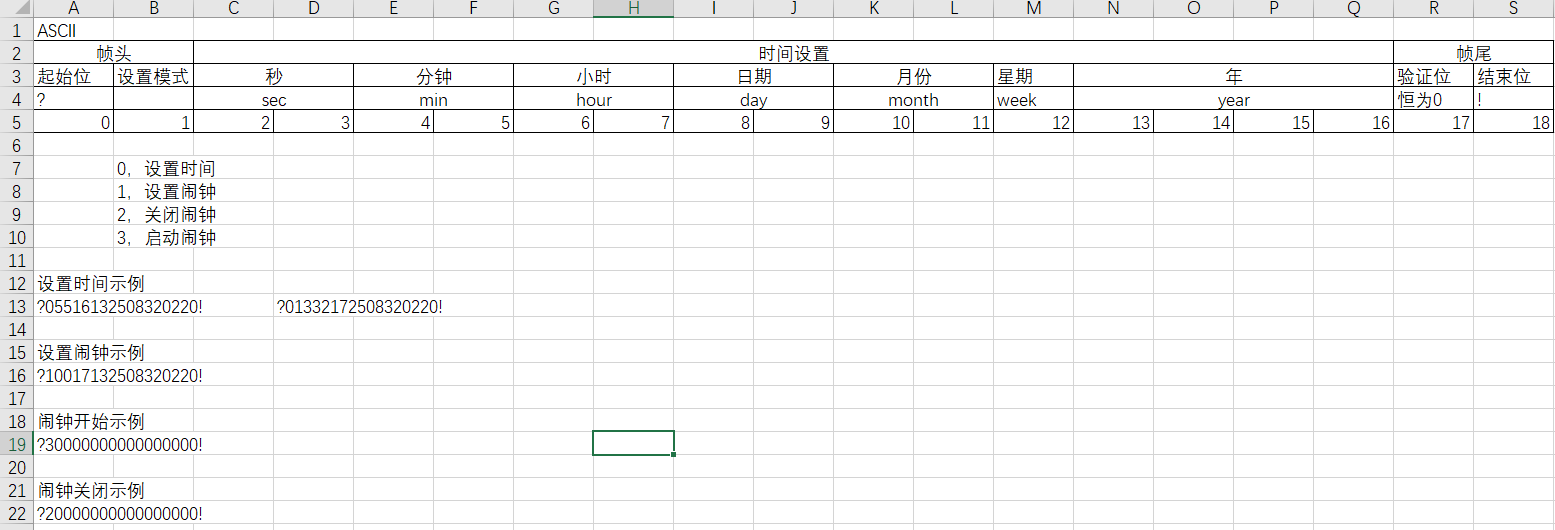
            }

            memset(CommandBuffer, 0, sizeof(CommandBuffer));

        }

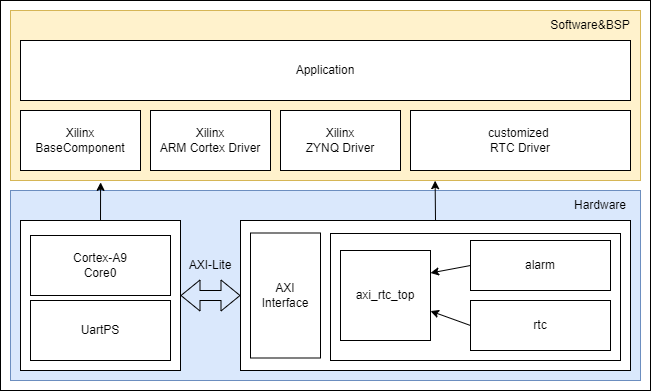
接收到指令数据后会根据以下帧结构对数据进行解析并赋值给rtc类的RTC设置属性或闹钟设置属性，并进行星期和年月日之间是否对应判断，如果二者不对应，则默认使用公式计算出的星期数据作为本次指令为星期赋值的数据，并以Error Week Input提示信息提醒操作者。如果二者对应则会直接设置正确的结果到RTC或闹钟。

如果输入的指令是2或3（关闭或开启闹钟），则会自动忽略后续数据位，直接进行开关闹钟的操作



【结果分析】

项目系统架构图如下所示：



本项目可以正确完成RTC项目基本要求中的1、2、3、4项基本功能，并且使用硬件设计RTC，还支持星期设置纠错功能、LED显示闹钟状态和按钮控制闹钟停止功能，满足实验中全部基本要求和扩展要求

